UNIVERSITÀ DI PISA

FACOLTÀ DI INGEGNERIA

CORSO DI LAUREA SPECIALISTICA IN INGEGNERIA INFORMATICA

Gestione degli utenti con LDAP per il controllo degli accessi basato su attributi

Primo Relatore: Prof. G. Dini

Secondo Relatore: Prof. C. Bernardeschi

Candidato: Luca Merloni

ANNO ACCADEMICO 2003–2004

Ai miei genitori,

ai miei nonni,

a mia sorella,

che mi hanno sostenuto

durante questi

5 anni

Ringraziamenti

Un ringraziamento speciale va a Nicola Provenzano per l'aiuto e i consigli forniti durante l'implementazione dell'applicazione, e ad Antonio Murgia per la collaborazione all'integrazione con la sua applicazione.

Indice

Elenco delle figure

Elenco delle tabelle

Introduzione

Lo sviluppo di un sistema per il controllo degli accessi, si basa sulla definizione di regole secondo cui è possibile consentire o negare l'accesso alle risorse, e sulla loro successiva implementazione come funzioni eseguibili da un sistema informatico. Tale processo di sviluppo è solitamente gestito seguendo un approccio di due fasi. Inizialmente si definisce la politica di sicurezza alla base del controllo degli accessi; poi si definiscono i meccanismi software e hardware che implementano la politica di sicurezza scelta.

Le politiche basate sui ruoli *Role-based Access Control* (*RBAC*) sono state introdotte nei primi anni '90 per la gestione del controllo nell'ambito di applicazioni commerciali. La motivazione principale dietro a questo tipo di politica è la necessità di specificare ed imporre politiche di controllo degli accessi strettamente legate alla struttura dell'organizzazione aziendale. Questa scelta è giustificata dal fatto che, in un numero consistente di organizzazioni, l'identità di un soggetto è rilevante solo dal punto di vista delle sue responsabilità civili o penali. Ai fini del controllo degli accessi, piuttosto che conoscere l'identità di un soggetto, è invece molto più importante conoscere i ruoli e le funzioni che tale soggetto svolge nell'ambito dell'organizzazione. La politica RBAC soddisfa tale esigenza assegnando i permessi di accesso ai ruoli ed assegnando questi ultimi agli utenti

in base alle loro responsabilità e qualifiche all'interno dell'organizzazione. Nel caso più generale, l'assegnazione dei permessi ai ruoli e dei ruoli agli utenti può essere modificata dinamicamente per soddisfare le mutate esigenze dell'organizzazione.
L'approccio RBAC può essere considerato come un caso particolare della politica di controllo degli accessi basata sugli attributi *Attributes-based Access Control* (*ABAC*), in cui la decisione di autorizzare o meno l'azione richiesta da un soggetto su di una particolare risorsa, dipende dalle caratteristiche, o attributi, del soggetto (*subject attributes*) nonché dagli attributi della risorsa stessa (*resource attributes*). Ne segue quindi che la politica ABAC generalizza la politica RBAC per quanto riguarda sia i soggetti sia le risorse. Mentre nella politica RBAC un soggetto è essenzialmente caratterizzato da due attributi predefiniti, l'identità ed il ruolo, nella politica ABAC l'insieme di attributi che caratterizza un soggetto non è definito a priori e può essere specificato in base alle esigenze del contesto applicativo.

L'oggetto di questa tesi è la realizzazione di un sistema per la gestione di utenti caratterizzati da attributi, attraverso l'utilizzo del protocollo *LDAP* (*Lightweight Directory Access Protocol*). Il sistema è stato realizzato come applicazione *enterprise*, utilizzando la piattaforma J2EE (*Java 2 Enterprise Edition*), un sistema operativo Linux e applicazioni open source come *OpenLDAP*. La scelta della piattaforma J2EE è dovuta al fatto che questo sistema deve essere integrato all'interno di un più ampio progetto di *Content Management* chiamato *CMOS* (*Content Management Open Source*), che utilizza già questa

tecnologia.
Sono stati affrontati diversi aspetti legati alla sicurezza in J2EE, come l'autenticazione e autorizzazione (controllo degli accessi all'EJB (*Enterprise Java Bean*) basato su ruoli), confidenzialità e integrità delle informazioni trasmesse e ricevute da OpenLDAP.

Nella relazione seguente si cerca di descrivere accuratamente il sistema, dai componenti realizzati alle scelte implementative effettuate, sino agli sviluppi futuri.

Nel capitolo **1** vengono trattate le tecnologie utilizzate e i concetti di naming e di directory, che sono alla base del protocollo *LDAP* (*Lightweight Directory Access Protocol*) e un esempio di uso per un controllo degli accessi basato su attributi.

Nel capitolo **2** vengono mostrate le specifiche e l'architettura del sistema. In particolare si specifica la modalità di installazione dei vari componenti.

Nel capitolo **3** vengono mostrate le configurazioni dei componenti del sistema per il funzionamento desiderato.

Nel capitolo **4** viene mostrata l'implementazione dello Stateless Session Bean *LdapServer* facendo riferimento alle interfacce Home e Remote.

Nel capitolo **5** viene mostrata l'implementazione del client stand-alone *LdapClientContext* per la gestione dei contesti nel server LDAP, e l'utilizzo del tool *ldapadd* messo a disposizione da *OpenLDAP*.

Nel capitolo **6** viene mostrata l'implementazione del client *LdapClientUsers*

per la gestione degli utenti nel server LDAP.

Nel capitolo **7** viene mostrata l'implementazione del client *LdapClientRoles* per la gestione dei ruoli nel server LDAP.

Nel capitolo **8** sono mostrati i risultati di alcuni test che sono stati eseguiti per fare misure sui tempi di risposta del sistema.

Nel capitolo **9** sono riportate le conclusioni.

Parte I

Sistema per la gestione degli utenti con LDAP

Capitolo 1

Introduzione al sistema

Il sistema di gestione degli utenti, per un controllo degli accessi basato su attributi, è stato realizzato utilizzando la piattaforma *J2EE* (J*ava 2 Enterprise Edition*) e il protocollo *LDAP* (*Lightweight Directory Access Protocol*). La piattaforma J2EE è stata utilizzata per sviluppare i componenti che implementano la logica business dell'applicazione, mentre il protocollo LDAP, permette di gestire gli attributi in formato X.500 all'interno di un repository.

1.1 Piattaforma J2EE

La piattaforma J2EE usa un modello distribuito a più livelli (*multitired*) per lo sviluppo di applicazioni *enterprise*. La logica dell'applicazione è divisa in *componenti* in base alle sue funzioni, e i vari componenti sono installati su macchine differenti, a seconda del livello dell'ambiente J2EE a cui appartengono (Figura **1.1**).

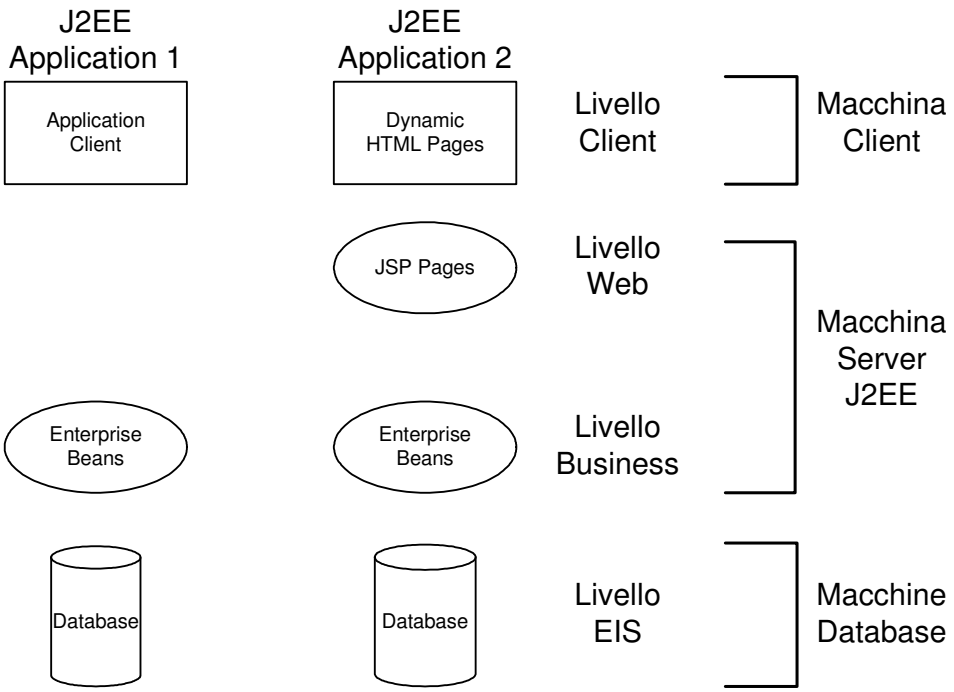

Figura 1.1: Applicazioni J2EE Multitired

La figura **1.1** mostra due applicazioni a più livelli, divise in livelli secondo la seguente lista:

- livello Client: componenti eseguiti sulla macchina client;
- livello Web: componenti eseguiti sul server J2EE;
- livello Business: componenti eseguiti sul server J2EE;
- livello Enterprise Information System (EIS): software presente sui server Database;

Sebbene un'applicazione J2EE può consistere di tre o quattro livelli, solitamente viene considerata di tre livelli perché i componenti sono distribuiti su tre locazioni differenti: macchine client, macchina con il server J2EE, macchine

con database.

Le applicazioni J2EE sono costituite da componenti. Un componente è un'unità software funzionale assemblata in un'applicazione J2EE con le sue classi, files e interconnessioni con gli altri componenti. Sono definiti i seguenti componenti J2EE:

- applicazioni client e applet sono componenti che vengono eseguiti sulla macchina client;
- Java Servlet e JavaServer Pages (JSP) sono componenti web che vengono eseguiti sul server;
- Enterprise JavaBeans (EJB) sono componenti business che vengono eseguiti sul server;

I componenti J2EE sono scritti nel linguaggio di programmazione Java e sono compilati nello stesso modo con cui si compila altri programmi Java. La differenza tra i componenti J2EE e le classi standard Java, è che i componenti J2EE sono assemblati in un'applicazione J2EE, sono verificati per essere conformi con le specifiche J2EE, e poi mandati in produzione *deployment*, dove vengono eseguiti e gestiti dal server J2EE.

Il codice business è gestito da *enterprise beans* eseguiti nel livello business. La figura **1.2** mostra come un enterprise bean riceve dati dal programma client, li processa (se necessario), e li invia al livello Enterprise Information System per memorizzarli. Un enterprise bean, inoltre, recupera i dati dall' EIS, li processa (se necessario), e li invia indietro al programma client.

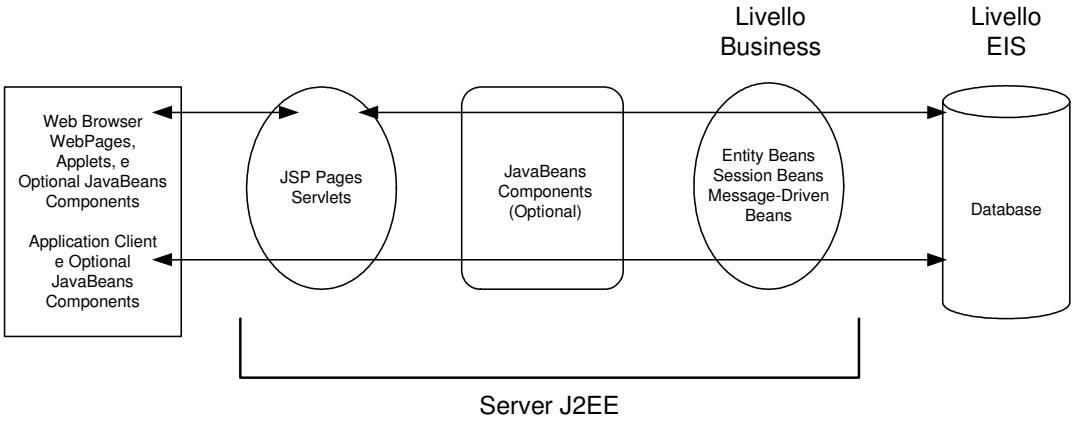

Figura 1.2: Livelli Business e EIS

Ci sono tre tipi di enterprise beans: *session beans*, *entity beans* e *message-driven beans*. Un session bean rappresenta una comunicazione temporanea con un client. Quando il client termina la sua esecuzione, il session bean e i suoi dati sono persi. L'entity bean, a differenza del session bean, rappresenta dei dati persistenti, memorizzati in una riga di una tabella del database. Se il client termina o il server viene chiuso, i servizi sottostanti assicurano che i dati dell'entity bean sono salvati. Un message-driven bean combina le features di un session bean e di un *Java Message Service (JMS) message listener*, permettendo così a un componente business di ricevere messaggi JMS in modo asincrono. Il fatto che J2EE sia indipendente dalla piattaforma, e che l'applicazione sia basata su un modello a più livelli, rende lo sviluppo di un'applicazione J2EE ancora più facile, grazie anche ai servizi sottostanti messi a disposizione per ciascun tipo di componente dal suo *container*. Il fatto di non sviluppare questi servizi, permette di concentrarsi solamente sulla soluzione al problema business. I containers (Figura **1.3**) sono interfacce tra un componente e le funzionalità a basso livello specifiche della piattaforma che supportano il componente.

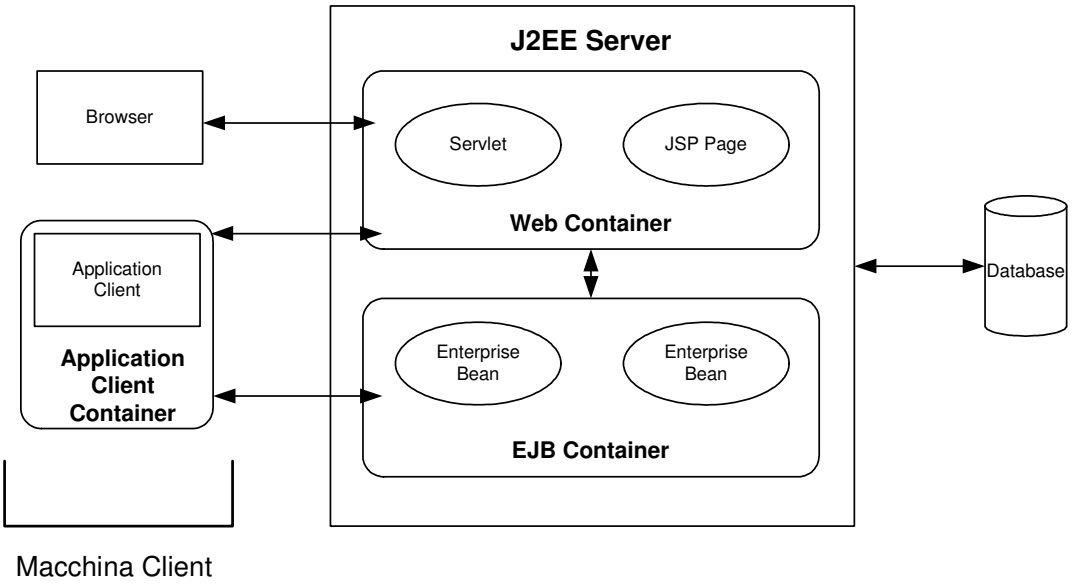

Figura 1.3: Server J2EE e containers

Un componente prima di essere eseguito, deve essere assemblato in un modulo J2EE e poi allocato nel suo container. Il processo di assemblaggio coinvolge i settaggi del container per ciascun componente dell'applicazione J2EE, e quelli dell'applicazione stessa. I settaggi del container permettono di configurare il supporto fornito dal server J2EE, includendo servizi come *sicurezza*, *gestione delle transazioni*, *Java Naming and Directory Interface* (*JNDI*) *lookup* e connettività remota *Remote Method Invocation* (*RMI*). Il modello della sicurezza in J2EE permette di configurare un componente web o un enterprise bean in modo tale che le risorse del sistema siano accedute solo dagli utenti autorizzati.

1.2 JAAS (Java Authentication and Authorization Service)

JAAS (*Java Authentication and Authorization Service*) consiste di un insieme di Java packages per l'autenticazione e autorizzazione dell'utente.
L'autenticazione in JAAS viene compiuta in modalità *pluggable*. Questo permette all'applicazione Java di rimanere indipendente dalle tecnologie sottostanti per l'autenticazione e permette al *security manager di JBossSX* di lavorare con infrastrutture di sicurezza differenti.
L'autenticazione consiste nel creare un istanza di un oggetto *LoginContext*, il quale necessita del nome di una *Configurazione* per determinare quale *LoginModule* verrà creato. Il LoginModule definisce la tecnologia per l'autenticazione, e utilizza *username* e *password* per effettuare l'autenticazione.
Per autenticare un soggetto sono seguiti i seguenti passi:

1. l'applicazione istanzia un LoginContext;
2. il LoginContext consulta una Configurazione per caricare il LoginModule configurato per l'applicazione;
3. l'applicazione invoca il metodo `login` del LoginContext;
4. il metodo `login` invoca il LoginModule caricato. Il LoginModule cerca di autenticare il soggetto. Se l'autenticazione ha successo, il LoginModule associa al soggetto l'username e password forniti;
5. il LoginContext ritorna lo stato dell'autenticazione all'applicazione;

6. se l'autenticazione ha avuto successo, l'applicazione può recuperare il soggetto autenticato dal LoginContext;

Il LoginContext consulta una Configurazione per determinare il LoginModule configurato per una particolare applicazione. Di conseguenza, è possibile utilizzare differenti LoginModules, senza cambiare l'applicazione stessa.

1.3 Servizi di Naming e Directory

Un servizio di naming permette di associare dei nomi ad oggetti e di poter effettuare una ricerca degli oggetti in base al loro nome. L'associazione di un nome con un oggetto è chiamata *binding*.
Per effettuare una ricerca di un oggetto in un sistema di naming, si deve fornire il nome dell'oggetto. Inoltre il sistema di naming specifica la sintassi che il nome deve avere. Questa sintassi è chiamata *convenzione sui nomi* (*naming convention*).
Si definisce *contesto* un insieme di associazioni nome-oggetto. A ciascun contesto è associato un naming convention. Un contesto fornisce l'operazione di *lookup*, per recuperare l'oggetto di cui è stato specificato il nome, ed altre operazioni per:

- associare un nome all'oggetto;
- rimuovere l'associazione tra un nome ed un oggetto;
- visualizzare i nomi associati agli oggetti;

All'interno del contesto un nome può essere associato ad un altro contesto chiamato *sotto contesto* o *subcontext*. Per esempio, nel file system UNIX la directory

`/usr` è un contesto, mentre la directory `/usr/bin` è un sotto contesto di `/usr`.
Molti servizi di naming sono estesi con *servizi di directory*. Un servizio di directory associa nomi ad oggetti, ma, inoltre, permette a questi oggetti di essere caratterizzati da una serie di *attributi*.
Un attributo è costituito da una coppia *nome* e *un insieme di valori*. La ricerca di un oggetto può avvenire così non soltanto in base al nome dell'oggetto, ma anche attraverso il valore che un particolare attributo assume.
Un servizio di directory, generalmente, permette anche di ricavare il valore degli attributi relativi all'oggetto specificato.
Gli oggetti gestiti da un sistema di directory sono detti *oggetti directory*. Un oggetto directory può rappresentare un qualunque elemento dell'ambiente operativo, ad esempio una stampante, un utente, un computer o una rete.
Un oggetto directory contiene gli attributi che descrivono l'oggetto che esso rappresenta. Per esempio un utente può essere rappresentato da un oggetto directory che ha come attributi il suo nome, cognome, userid e password.
Una *directory* è un insieme di oggetti directory.
Infine un servizio di directory è un servizio che fornisce operazioni per aggiungere, rimuovere, modificare e cercare il valore degli attributi associati all'oggetto in una directory. Generalmente è ottimizzato per fornire una risposta veloce a ricerche di grossi volumi di informazioni.

1.4 LDAP (Lightweight Directory Access Protocol)

LDAP (*Lightweight Directory Access Protocol*) è un protocollo leggero per accedere a servizi di directory (basati sullo standard X.500) o a un servizio standalone. *Leggero* perché rispetto al protocollo DAP, LDAP utilizza un qualsiasi protocollo di trasporto come TCP, evitando l'overhead introdotto dai livelli *session/presentation*. LDAP è basato su un modello client-server nel quale il client crea una connessione TCP a un server LDAP, attraverso la quale invia richieste e riceve risposte.

In LDAP il modello delle informazioni è basato su *entry* che corrispondono agli oggetti directory visti precedentemente **§1.3**. Un'entry è una collezione di attributi identificata da un *Distinguished Name* (*DN*). Ciascun attributo è costituito da un nome e da uno o più valori. I nomi sono nel formato X.500, cioè stringhe come `cn` per *common name* o `mail` per *email*. La sintassi del valore dipende dal tipo dell'attributo. Per esempio l'attributo `cn` potrebbe assumere come valore *Mario Rossi*, l'attributo `mail` il valore *marior@myserver.com*.

In LDAP le entry sono organizzate secondo una struttura gerarchica ad albero chiamata *Directory Information Tree* (*DIT*). La root dell'albero rappresenta il contesto top della gerarchia, mentre a scendere troviamo i vari sotto contesti. Le foglie rappresentano le entry finali che fanno parte del sotto contesto e sono identificate in modo univoco dal Distinguished Name che viene creato seguendo il percorso dalla foglia fino alla radice. Ad esempio, con riferimento alla Figura **1.4**, il Distinguished Name per l'entry *lucam* è dato da:

uid=lucam,ou=People,o=myserver.com.

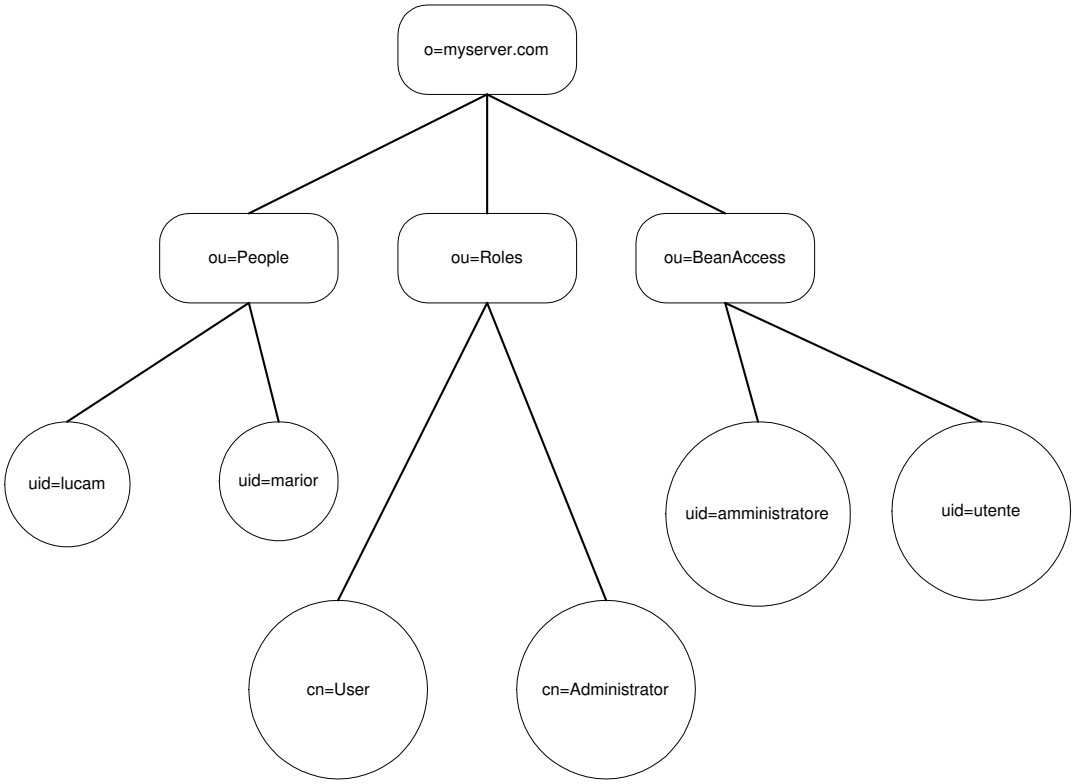

Figura 1.4: Un semplice esempio di Directory Information Tree

Gli attributi che caratterizzano l'entry *lucam* possono essere:

- commonName (`cn`): solitamente nome e cognome;
- surname (`sn`): cognome;
- userID (`uid`): userid univoco nel sotto contesto;
- email (`mail`): indirizzo di posta elettronica;

LDAP permette, in più, di controllare quali attributi sono obbligatori e quali sono facoltativi in un'entry, attraverso l'uso di uno speciale attributo chiamato `objectClass`. Il valore dell'attributo `objectClass` determina le regole cui

l'entry deve ubbidire.
LDAP definisce le operazioni per interrogare e aggiornare la directory. Queste operazioni permettono di:

- aggiungere una nuova entry alla directory;
- cancellare un'entry esistente dalla directory;
- modificare un'entry esistente: cambiare gli attributi che devono essere presenti oppure cambiare il valore degli attributi;
- cambiare il nome con cui è avvenuto il bind di un'entry;

Le operazioni di ricerca permettono di recuperare porzioni della directory che soddisfano i criteri specificati nella richiesta. Per esempio è possibile cercare all'interno del sotto albero *o=myserver.com* (Figura **1.4**), tutte le persone con l'attributo `commonName` che assume come valore Mario Rossi, e recuperare per ciascuna il valore dell'indirizzo email.
Alcuni servizi di directory non forniscono meccanismi di protezione delle informazioni contro accessi non autorizzati. LDAP, invece, dà la possibilità al client di autenticarsi, o di fornire la sua identità al server di directory. Inoltre offre supporto per la confidenzialità ed integrità delle informazioni memorizzate.
LDAP si basa sul modello client-server. Uno o più LDAP server contengono i dati che costituiscono il Directory Information Tree. Il client si connette al server, fa la richiesta e il server risponde con i dati e/o con il puntatore alla locazione dove può recuperare informazioni addizionali (tipicamente un altro server LDAP). Indipendentemente dal server a cui si collega, il client ha sempre la stessa visione dell'albero.

1.5 Esempio di un controllo degli accessi basato su attributi dell'utente

Consideriamo un servizio web attraverso il quale è possibile ascoltare in streaming brani musicali di artisti. Abbiamo a disposizione un insieme di risorse (i brani musicali) e un insieme di utenti registrati, i quali vogliono compiere delle operazioni sulle risorse (ascoltare le canzoni in streaming). È necessario quindi creare un servizio di controllo degli accessi in modo tale da poter verificare la richiesta del soggetto, e controllare se ha la possibilità di compiere una particolare azione su una particolare risorsa. Ciascun soggetto è un'entry all'interno del server LDAP, ed è caratterizzato da un insieme di attributi:

Attributi del soggetto	
cn	nome e cognome
givenName	nome
sn	cognome
uid	userid
userPassword	password
ou	organizationalUnit
mail	indirizzo email
objectClass	objectClass in cui sono specificate le regole che l'entry deve ubbidire
credito	credito disponibile
banda	banda in download espressa in Kb
generePreferito	genere musicale preferito
strumentoMusicale	strumento musicale

Tabella 1.1: Attributi del soggetto

Ipotizziamo che i brani siano disponibili con diversi bitrate come 56Kb (scarsa qualità), 128Kb (buona qualità) e 256Kb (ottima qualità). Supponiamo inoltre

che il costo dell'ascolto del brano sia proporzionale alla qualità scelta:

$$56Kb \rightarrow 1\ euro$$

$$128Kb \rightarrow 2\ euro$$

$$256Kb \rightarrow 4\ euro$$

Un soggetto caratterizzato da una banda di 128Kb e da un credito di 10 euro, decide di ascoltare un brano musicale caratterizzato da un bitrate di 56Kb: la richiesta avviene con successo.
Se il solito soggetto decide di ascoltare una canzone caratterizzata da un bitrate di 256Kb la richiesta viene negata, in quanto la banda del soggetto risulta insufficiente per un corretto ascolto.
Il nostro compito è quello di implementare un sistema per la gestione degli utenti caratterizzati da attributi attraverso il protocollo LDAP, e fornire i valori degli attributi di un soggetto, qualora ne sia fatta richiesta al sistema.

Parte II

Specifiche di progetto ed implementazione

Capitolo 2

Specifiche e architettura del sistema

2.1 Specifica dei requisiti del sistema

Il sistema deve fornire un tool completo per la gestione degli *utenti*, dei *ruoli* e dei *contesti* all'interno del server LDAP. In particolare deve permettere di poter assegnare un utente ad un particolare ruolo, in modo da poter decidere quali operazioni sono permesse all'utente.

Per quanto riguarda la sicurezza del sistema, è richiesto che il valore dell'attributo *password* di un'entry dell'utente, non sia memorizzato in chiaro all'interno del server LDAP. Lo scambio delle informazioni tra client e server LDAP deve avvenire attraverso un canale sicuro come *SSL*, per garantire *confidenzialità*, *integrità* delle informazioni e *autenticità* del server LDAP e del client che si connette.

È inoltre richiesto, che l'accesso al server LDAP non degradi le prestazioni del

sistema da un punto di vista dei tempi della risposta. È necessario quindi implementare un meccanismo che renda *scalabile* il sistema all'aumentare delle richieste al server LDAP.

2.2 Architettura e installazione del sistema

2.2.1 Architettura del sistema

Per implementare il sistema visto nell'esempio **§1.5**, sono stati scelti i seguenti componenti:

- sistema operativo open source Linux;
- Java 2 Standard Edition http://java.sun.com/ (versione 1.4.2_06);
- OpenLDAP (versione 2.2.18) http://www.openldap.org/: implementazione open source del protocollo LDAP;
- BerkeleyDB (versione 4.2.52.NC) http://www.sleepycat.com/: database utilizzato da OpenLDAP per la memorizzazione delle entry;
- OpenSSL (versione 0.9.7e) http://www.openssl.org/: implementazione open source del protocollo SSL;
- JBoss (versione 3.2.5) http://www.jboss.org/: Application Server;

Il client che interagisce con OpenLDAP è realizzato come componente *Enterprise JavaBean* (*EJB*) **§1.1**. Il vantaggio di questa scelta consiste nell'usufruire del supporto ad un ambiente transazionale, sicuro e protetto offerto dall'*Application*

Server.

OpenSSL ci permette di implementare un canale sicuro *SSL* tra il client e il server OpenLDAP, in modo da garantire autenticità del server e del client, confidenzialità e integrità delle informazioni trasmesse.

OpenLDAP mette a disposizione il server *slapd* ed un insieme di strumenti lato-client che permettono di interagire con slapd. In fase di installazione è possibile specificare il database di back-end desiderato (*BerkeleyDB*, *ldbm* o altro).

La scelta del database BerkeleyDB è motivata dal fatto che questo database è progettato appositamente per applicazioni scalabili, ad alte performance ed orientate alle transazioni. BerkeleyDB mette a disposizione una libreria di funzioni per l'accesso ai dati e la loro gestione. Questo permette a BerkeleyDB di operare direttamente all'interno dello spazio di indirizzamento dell'applicazione che lo usa (nel nostro caso OpenLDAP). Non c'è server separato da installare o da amministrare. L'applicazione client fa una semplice chiamata alla libreria per memorizzare, recuperare e modificare i dati, che sono residenti in un file sul file-system locale.

In figura **2.1** viene mostrata l'architettura del sistema.

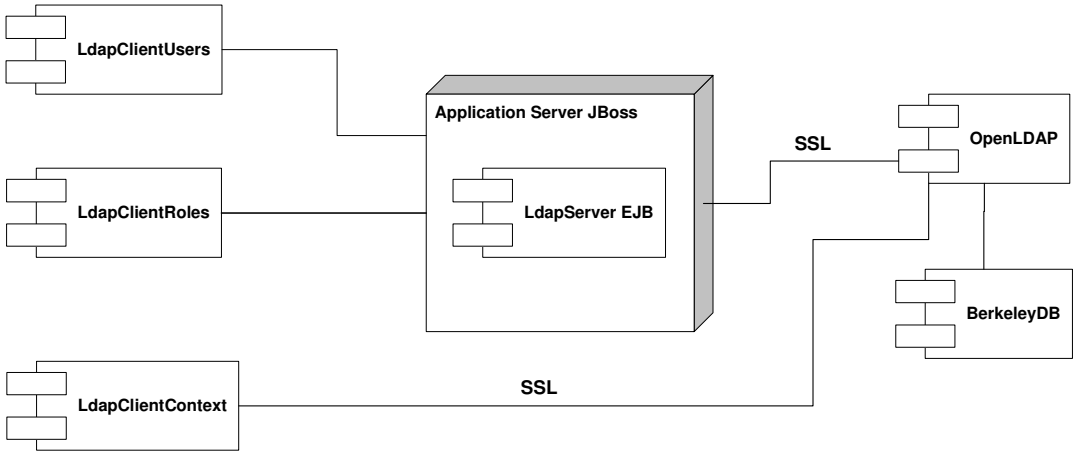

Figura 2.1: Architettura del sistema

2.2.2 Installazione di JBoss

Per installare l'application server JBoss è necessario disporre di un PC con la Java 2 Standard Edition già installata. Recuperare il pacchetto di installazione dal sito ufficiale *www.jboss.org* ed estrarlo per semplicità nella directory `Home` con il comando:

```
tar xzvf jboss-3.2.5.tar.gz
```

È inoltre necessario settare il valore delle seguenti variabili di ambiente nel file `/etc/profile` come utente `root`:

```
export JBOSS_HOME="/home/luca/jboss-3.2.5";
export JAVA_HOME="/home/luca/j2sdk1.4.2_06";
export CLASSPATH="/home/luca/j2sdk1.4.2_06/lib/tools.jar:
                  /home/luca/j2sdk1.4.2_06/jre/lib/ext/jaas.jar";
export LD_LIBRARY_PATH="/usr/local/BerkeleyDB.4.2/lib";
```

La prima linea serve per settare il path dove è stato estratto il pacchetto di JBoss, mentre la seconda linea specifica il path di installazione della Java 2

Standard Edition. La terza riga specifica i jar da includere al classpath. In particolare, il secondo jar è necessario per *JAAS* (*Java Authentication and Authorization Service*) **§1.2**, utilizzato per l'autenticazione del client sull' EJB. L'ultima riga specifica il path delle librerie per il database BerkeleyDB.
Per mandare in esecuzione il server JBoss, basta andare nella directory `/home/luca/jboss-3.2.5/bin/` ed eseguire il comando:

```
./run.sh
```

Il server si metterà in ascolto per tutti i servizi sull'indirizzo localhost (127.0.0.1). Per specificare un indirizzo diverso da quello di default, lanciare il server con il comando

```
./run.sh -b 192.168.0.1
```

dove 192.168.0.1 è il nuovo indirizzo.

2.2.3 Installazione di OpenSSL

Recuperare il pacchetto dei sorgenti dal sito ufficiale *www.openssl.org* ed estrarlo nella directory `Home` con il comando:

```
tar xzvf openssl-0.9.7e.tar.gz
```

Per l'installazione spostarsi nella directory appena creata `openssl-0.9.7e/` ed eseguire i seguenti comandi:

```
./config
make
make test
make install
```

2.2.4 Installazione di BerkeleyDB

Recuperare il pacchetto dei sorgenti dal sito ufficiale *www.sleepycat.com* ed estrarlo nella directory `Home` con il comando:

```
tar xzvf db-4.2.52.NC.tar.gz
```

Spostarsi nella directory `db-4.2.52.NC/build_unix/` ed eseguire i seguenti comandi:

```
../dist/configure
make
make install
```

2.2.5 Installazione di OpenLDAP

Recuperare il pacchetto dei sorgenti dal sito ufficiale *www.openldap.org* ed estrarlo nella directory `Home` con il comando:

```
tar xzvf openldap-2.2.18.tar.gz
```

Spostarsi nella directory appena creata `openldap-2.2.18/` ed eseguire i seguenti comandi:

```
LD_LIBRARY_PATH="/usr/local/BerkeleyDB.4.2/lib" \
env CPPFLAGS="-I/usr/local/BerkeleyDB.4.2/include" \
LDFLAGS="-L/usr/local/BerkeleyDB.4.2/lib" \
./configure \
--with-bdb=/usr/local/BerkeleyDB.4.2 \
--with-bdb-libdir=/usr/local/BerkeleyDB.4.2/lib \
--with-bdb-incdir=/usr/local/BerkeleyDB.4.2/include \
--with-tls \
--enable-ssl=yes \
--with-ssl=/home/luca/openssl-0.9.7e

make depend
make
```

```
make test
make install
```

È importante verificare che durante la fase di *./configure* siano state riconosciute le librerie di OpenSSL e BerkeleyDB.

Per eseguire il server slapd abilitandolo sulla porta 389 e 636 (SSL) eseguire:

```
/usr/local/libexec/slapd -d127 -h "ldap:/// ldaps:///"
```

Con l'opzione -d si specifica il livello di debug. Per terminare il server basta digitare `Ctrl+c` sulla console dove è in esecuzione o il seguente comando:

```
kill -INT `cat /usr/local/var/run/slapd.pid`
```

Il path del file *slapd.pid* deve essere lo stesso specificato nel file di configurazione *slapd.conf* **§3.2**.

Capitolo 3

Configurazione del sistema

3.1 Configurazione di JBoss

3.1.1 LdapLoginModule

JAAS **§1.2** ci permette di fornire un controllo degli accessi per l'EJB basato su ruoli. Questa tecnologia ci permette di decidere chi può accedere all'EJB e quali metodi autorizzare in base al ruolo di chi si autentica. È stato deciso di gestire i ruoli all'interno del server LDAP. Il LoginModule *LdapLoginModule* ci permette di effettuare il controllo degli accessi basandosi sui ruoli creati nel server LDAP.

È necessario inserire la Configurazione del LoginModule nel file *auth.conf* nella directory `/home/luca/jboss-3.2.5/client/`:

```
ldap {
    org.jboss.security.ClientLoginModule required;
    org.jboss.security.auth.spi.LdapLoginModule required
    java.naming.factory.initial="com.sun.jndi.ldap.LdapCtxFactory"
    java.naming.provider.url="ldaps://127.0.0.1/"
```

```
    java.naming.security.authentication="simple"
    java.naming.security.principal="cn=Manager,o=myserver.com"
    java.naming.security.credentials="openldap"
    principalDNPrefix="uid="
    uidAttributeID="description"
    roleAttributeID="l"
    principalDNSuffix=",ou=BeanAccess,o=myserver.com"
    roleCtxDN="ou=Roles,o=myserver.com";
};
```

È necessario fornire un identificatore alla Configurazione, in questo caso è stato scelto *ldap*. All'interno della Configurazione viene richiesto l'utilizzo del LoginModule *org.jboss.security.ClientLoginModule* per richiedere le credenziali al soggetto, il LoginModule *org.jboss.security.auth.spi.LdapLoginModule* per effettuare l'autenticazione e autorizzazione del soggetto utilizzando il server LDAP. Le opzioni specificate per il modulo *LdapLoginModule* sono:

- java.naming.factory.initial: il contesto iniziale JNDI;
- java.naming.provider.url: url del server LDAP. Come è mostrato nella Configurazione, se il valore dell'url presenta la `s` finale, specifichiamo che il collegamento con il server LDAP deve avvenire attraverso SSL;
- java.naming.security.authentication: tipo di autenticazione richiesta dal server LDAP;
- java.naming.security.principal: root directory userid (lo stesso del file di configurazione di OpenLDAP *slapd.conf* **§3.2**);
- java.naming.security.credentials: root directory password **§3.2**;

- passwordValidationMethod: specifica in che modo il LoginModule effettua la validazione della password. Il valore Cams permette che la verifica avvenga utilizzando una funzione Hash (MD5, SHA);
- principalDNPrefix: attributo del Relative Distinguished Name che identifica l'utente nel subContext degli utenti che hanno accesso all'EJB + '=';
- uidAttributeID: nome dell'attributo utilizzato per l'userid dell'utente nell'entry del ruolo;
- roleAttributeID: nome dell'attributo che identifica il ruolo, utilizzato nell'entry del ruolo;
- principalDNSuffix: ',' + Distinguished Name che identifica il subContext degli utenti che hanno accesso all'EJB;
- rolesCtxDN: Distinguished Name del subContext dei Ruoli;

Per l'opzione *java.naming.security.authentication* è possibile specificare i seguenti valori:

- none: anonymous;
- simple: sono richieste username e password in chiaro;
- sasl_mech: è possibile utilizzare meccanismi *SASL* (*Simple Authentication and Security Layer*) come Digest-MD5, External, Kerberos V5;

Scegliendo il valore simple, nel caso in cui non vengano forniti username e password, l'utente viene considerato *anonymous*. Se viene specificato solo l'username, l'utente viene invece considerato *unauthenticated*, mentre se vengono

forniti username e password l'utente è *authenticated*. L'autenticazione avviene verificando la presenza dell'entry relativa all'utente, nel subContext degli utenti che hanno accesso all'EJB. L'utente deve essere identificato dal Distinguished Name:

```
principalDNPrefix + username fornito da console + principalDNSuffix
```

Se l'utente esiste, guarda nel subContext dei ruoli, in quale ruolo l'utente deve essere mappato, verificando la presenza dell'username tra i valori dell'attributo specificato nell'opzione *uidAttributeID*. L'utente, dopo essersi autenticato, può avere l'autorizzazione di accedere ad un particolare metodo dell'EJB, a seconda di quanto specificato nel *Deployment Descriptor* dell'EJB **§4.5**.

3.1.2 Pooling di Stateless Session Bean

È possibile configurare JBoss in modo tale da avere un *Pool* di istanze di Stateless Session Bean tutte equivalenti per il *Container*. In questo modo, in caso di grosso numero di richieste, l'EJB non si comporta come un unico punto di accesso al server LDAP, perché le richieste vengono servite da più istanze equivalenti, senza compromettere così le prestazioni del sistema.
Il file di configurazione di JBoss da modificare è situato nella directory `/home/luca/jboss-3.2.5/server/default/conf/` e si chiama *standardjboss.xml*. Inserire le seguenti righe nella configurazione del container per lo Standard Stateless Session Bean:

```
<container-pool-conf>
 <MinimumSize>2</MinimumSize>
 <MaximumSize>10</MaximumSize>
</container-pool-conf>
```

Specifichiamo così il numero minimo e massimo delle istanze all'interno del Pool. Un'istanza verrà rimossa solo quando ritenuto necessario dal container (poche richieste in arrivo, poca memoria disponibile).

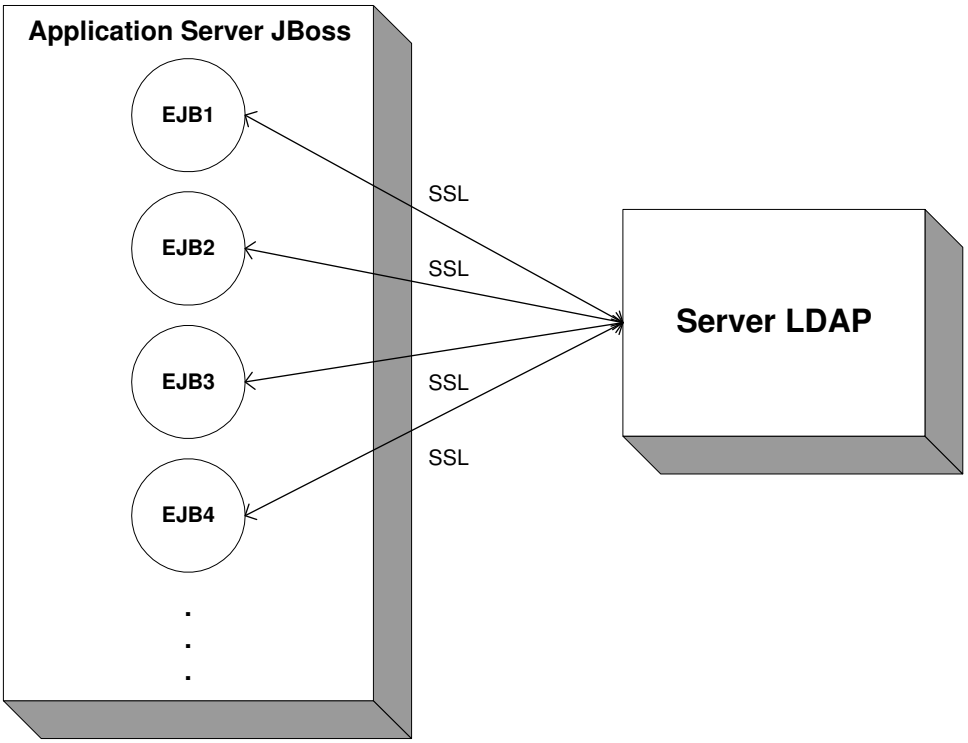

Figura 3.1: Pooling di EJB

Utilizzando un Pool di Stateless Session Bean, riusciamo ad ottenere un *connection pooling* di connessioni SSL tra l'Application Server e il server LDAP. Al momento della creazione di ogni istanza, viene stabilita una connessione SSL col server LDAP, e viene chiusa solo quando il container decide di rimuovere l'istanza dell'EJB. All'inizio viene creato un numero minimo di istanze, come specificato nel file *standardjboss.xml*, e quindi le richieste dei client verranno soddisfatte utilizzando le connessioni esistenti. Se il numero delle richieste aumenta, il container può decidere di creare altre istanze di EJB, e quindi stabilire

altre connessioni col server LDAP, smistando così le richieste anche su queste nuove connessioni.
Al momento non sembra essere disponibile un *connettore JCA* per OpenLDAP, capace di implementare un vero connection pooling tra Application Server e Database, cosa disponibile per altri tipi di backend come *MySql* e *Oracle*.

3.2 Configurazione di OpenLDAP

Per configurare OpenLDAP è necessario modificare il file *slapd.conf* nella directory `/usr/local/etc/openldap/`. All'interno del file è possibile specificare tre tipi di informazioni per la configurazione:

- globali;
- specifiche di backend;
- specifiche del database;

Le informazioni globali sono le prime a essere specificate, seguite da informazioni associate ad un particolare tipo di backend e da informazioni associate a una particolare istanza di database. Le direttive globali possono essere sovra scritte dalle direttive di backend e/o dalle direttive del database, mentre le direttive di backend possono essere sovra scritte dalle direttive del database. Le direttive globali si applicano a tutti i backend e database. Tra le direttive globali è possibile includere i file schema, nel nostro caso sono utili i seguenti file:

```
include         /usr/local/etc/openldap/schema/core.schema
include         /usr/local/etc/openldap/schema/inetorgperson.schema
include         /usr/local/etc/openldap/schema/musica.schema
```

È possibile creare un Access List con la quale si specifica *chi* ha accesso a *cosa*. Le parti principali sono:

- <what>: indica le entry e/o gli attributi ai quali il controllo deve essere applicato;
- <who>: le entità che hanno accesso alla risorsa;
- <access>: specifica il tipo di accesso consentito;

Nella seguente tabella sono riportate le possibili entità che possono accedere alle risorse:

<who>	**Entità**
*	Tutti, incluso anonymous e utenti non autenticati
anonymous	Anonymous
users	Utenti autenticati
self	Utente associato con l'entry da controllare
dn[.<basic-style>]=<regex>	Utenti che soddisfano un'espressione regolare
dn.<scope-style>=<DN>	Utenti che soddisfano un Distinguished Name

Tabella 3.1: Valori per <who>

Un esempio di Access List è:

```
access to *
       by self write
       by anonymous auth
       by * read
```

Questa direttiva permette agli utenti di modificare le proprie entry, agli utenti anonymous di autenticarsi e a tutti gli altri utenti di accedere in sola lettura alle entry. Solo il primo *by <who>* verrà applicato in caso di match, di conseguenza all'utente anonymous è permesso di autenticarsi e non di accedere in lettura.

Altre direttive globali che si devono inserire, sono quelle che riguardano i certificati X.509 del server LDAP **§3.4**:

```
TLSCipherSuite HIGH:MEDIUM:+SSLv2
TLSCACertificateFile /home/luca/servercer/cacert.pem
TLSCertificateFile /home/luca/servercer/servercrt.pem
TLSCertificateKeyFile /home/luca/servercer/serverkey.pem
```

Con la prima riga si specifica quali suite di cifratura sono accettate. Con le altre tre righe si specificano i path dei certificati relativi alla Certification Authority, al server LDAP, e il path del file contenente la chiave privata del server.
Per quanto riguarda le direttive del database, queste vengono applicate solo allo specifico tipo di database dichiarato:

```
database        bdb
suffix          "o=myserver.com"
rootdn          "cn=Manager,o=myserver.com"
rootpw          {SSHA}BWES0opBciUWTdCFaBShk3KuQd7oJUs4
directory       /usr/local/var/openldap-data
schemacheck     off
index   objectClass     eq
```

Con la direttiva *database* si specifica il tipo di database da utilizzare. Nella seguente tabella sono riportati alcuni possibili database:

Tipi	**Descrizione**
bdb	Berkeley DB transactional database
ldbm	Lightweight DBM database
sql	SQL Programmable database

Tabella 3.2: Possibili database per OpenLDAP

La direttiva *suffix* specifica il suffisso del Distinguished Name delle query che vengono passate al database. La direttiva *rootdn* specifica il Distinguished Name che non è soggetto a controllo degli accessi o limiti amministrativi per le

operazioni sul database. Non è necessario che questo Distinguished Name si riferisca a un'entry all'interno del server LDAP. Con la direttiva *rootpw* specifichiamo una password per il Distinguished Name rootdn. È consigliabile, per ragioni di sicurezza, inserire l'Hash della password e non la password in chiaro. Per generare l'Hash della password utilizzare il tool *slappasswd* messo a disposizione da OpenLDAP:

```
slappasswd -h {SSHA} -s openldap
```

Con l'opzione -h si specifica la funzione Hash da utilizzare. Nella seguente tabella sono riportate le possibili funzioni Hash:

Funzione Hash	**Descrizione**
CRYPT	
MD5	Digest MD5
SMD5	Digest MD5 con salt
SHA	
SSHA	SHA con salt

Tabella 3.3: Possibili funzioni Hash per la password

Con la direttiva *directory* si specifica il path della directory contenente i file del database e gli indici. Con la direttiva *schemacheck* è possibile forzare il controllo dello schema **§3.3** per ogni inserimento di una nuova entry, specificando come valore *on*. Con il valore *off* lo schema non viene controllato.
La direttiva *index* specifica gli indici da mantenere per il dato attributo. Nel nostro caso stabiliamo un indice di tipo *equality* sull'attributo objectClass. Per default nessun indice viene mantenuto, ma è consigliato averne uno di tipo equality almeno sull'attributo objectClass.

3.3 Creazione dello schema

OpenLDAP mette a disposizione un certo numero di schemi standard che possono essere utilizzati per definire gli attributi che caratterizzano il soggetto. Nel nostro esempio abbiamo la necessità di creare un nuovo schema chiamato *musica*, all'interno del quale possiamo definire nuovi attributi ed estendere così gli schemi esistenti messi a disposizione da OpenLDAP.

Lo schema musica è un file chiamato *musica.schema*, e viene creato, per semplicità, all'interno della directory `/usr/local/etc/openldap/schema/` e il path deve coincidere con quello specificato nel file di configurazione *slapd.conf* **§3.2**. Nel file verranno creati nuovi attributi e un nuovo objectClass.

Ciascun elemento dello schema è identificato univocamente attraverso un *Object Identifier* (*OID*). Questi identificatori sono gerarchici, quindi è necessario assegnarli correttamente ai vari elementi dello schema. Per esempio nel nostro caso possiamo assegnare agli attributi un OID del tipo 1.1.x dove x varia da 1 in poi, mentre all'objectClass possiamo assegnare l'OID 1.2.1

È possibile registrare l'OID (se non già esistente) presso la *Internet Assigned Numbers Authority* (*IANA*) a costo nullo. Basta riempire una form all'indirizzo *http://www.iana.org/cgi-bin/enterprise.pl* e in pochi giorni è possibile ricevere l'OID di base, del tipo 1.3.6.1.4.1.X

È necessario assegnare a ciscun elemento dello schema oltre all'OID, almeno un nome testuale. Il nome deve essere descrittivo e non deve coincidere con quelli già esistenti negli altri schemi.

La direttiva *attributetype* è utilizzata per definire un nuovo attributo. I nuovi attributi per lo schema musica sono:

```
attributetype ( 1.1.1
        NAME 'generePreferito'
        EQUALITY caseIgnoreMatch
        SUBSTR caseIgnoreSubstringsMatch
        SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{1024} )

attributetype ( 1.1.2
        NAME 'credito'
        DESC 'credito residuo per fare l'acquisto della canzone'
        EQUALITY caseIgnoreMatch
        SUBSTR caseIgnoreSubstringsMatch
        SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{1024} )

attributetype ( 1.1.3
        NAME 'banda'
        DESC 'àvelocit in DownLink dell'utente'
        EQUALITY caseIgnoreMatch
        SUBSTR caseIgnoreSubstringsMatch
        SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{1024} )

attributetype ( 1.1.4
        NAME 'strumentoMusicale'
        EQUALITY caseIgnoreMatch
        SUBSTR caseIgnoreSubstringsMatch
        SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{1024} )
```

Sotto è riportata la tabella degli OID delle sintassi più comuni:

Name	**OID**	**Descrizione**
boolean	1.3.6.1.4.1.1466.115.121.1.7	boolean value
directoryString	1.3.6.1.4.1.1466.115.121.1.15	Unicode (UTF-8) string
distinguishedName	1.3.6.1.4.1.1466.115.121.1.12	LDAP DN
integer	1.3.6.1.4.1.1466.115.121.1.27	integer
numericString	1.3.6.1.4.1.1466.115.121.1.36	numeric string
OID	1.3.6.1.4.1.1466.115.121.1.38	object identifier
octetString	1.3.6.1.4.1.1466.115.121.1.40	arbitary octets

Tabella 3.4: OID delle sintassi più usate

1024 rappresenta la lunghezza massima della stringa. Sono specificate anche le regole di matching nel caso di *uguaglianza* e *sotto stringa*. Sotto è riportata la tabella per le regole di matching più comunemente usate:

Name	**Tipo**	**Descrizione**
booleanMatch	equality	boolean
caseIgnoreMatch	equality	case insensitive, space insensitive
caseIgnoreOrderingMatch	ordering	case insensitive, space insensitive
caseIgnoreSubstringsMatch	substrings	case insensitive, space insensitive
caseExactMatch	equality	case sensitive, space insensitive
caseExactOrderingMatch	ordering	case sensitive, space insensitive
caseExactSubstringsMatch	substrings	case sensitive, space insensitive
distinguishedNameMatch	equality	distinguished name
integerMatch	equality	integer
integerOrderingMatch	ordering	integer
numericStringMatch	equality	numerical
numericStringOrderingMatch	ordering	numerical
numericStringSubstringsMatch	substrings	numerical
octetStringMatch	equality	octet string
octetStringOrderingStringMatch	ordering	octet string
octetStringSubstringsStringMatch	ordering	octet string
objectIdentiferMatch	equality	object identifier

Tabella 3.5: Regole di matching più usate

La direttiva *objectclass* è usata per definire un nuovo objectClass. Nel nostro caso è stato creato l'objectClass `musica`:

```
objectclass ( 1.2.1
        NAME 'musica'
        DESC 'utente in musica'
        SUP inetOrgPerson
```

```
MUST ( cn $ sn $ uid $ userPassword)
MAY ( givenName $ ou $ mail $ genere Preferito
        $ credito $ banda $ strumentoMusicale) )
```

L'objectClass permette di specificare le regole cui l'entry nel server LDAP dovrà ubbidire. In particolare è possibile specificare con MUST gli attributi obbligatori e con MAY quelli opzionali. Inoltre con SUP si specifica che l'objectClass musica, eredita le regole dell'objectClass inetOrgPerson, e quindi, quali attributi devono essere obbligatori e opzionali.
Nel nostro esempio gli attributi `cn`, `sn`, `uid` e `userPassword` devono essere presenti in un'entry del server LDAP.

3.4 Generazione dei certificati X.509

Per stabilire un canale sicuro tra il server LDAP e l'EJB, è necessario generare i certificati, per far si che il protocollo SSL vada a buon fine, cioè dotare il server LDAP di un certificato nel formato X.509.
È stato scelta la versione di SSL in cui è richiesta l'autenticazione solo del server LDAP, in quanto il client passa le sue credenziali secondo una delle modalità di autenticazione previste dal server **§3.1.1**.
Conviene creare la directory sotto la `Home` nella quale verrà poi inserito il certificato generato, per es. `servercer/`.

3.4.1 Creazione della Certification Authority

Se esiste la possibilità di accedere ad una *Certification Authority* fidata, procedere per recuperare il certificato del server LDAP. Se non è disponibile accedere

a una Certification Authority fidata, OpenSSL permette di arrivare allo stesso risultato in modo facile e veloce.
Spostarsi nella directory `/usr/local/ssl/misc/` ed eseguire i comandi per la creazione di una nuova Certification Authority:

```
./CA.sh -newca
```

Viene richiesto l'inserimento di una password che verrà utilizzata ogni volta sarà necessario accedere alla Certification Authority. Durante la procedura è richiesto l'inserimento di un valore per i seguenti campi:

- Country Name (2 letter code) [AU];
- State or Province Name (full name) [Some-State];
- Locality Name (eg, city) [];
- Organization Name (eg, company) [Internet Widgits Pty Ltd];
- Organizational Unit Name (eg, section) [];
- Common Name (eg, YOUR name) [];
- Email Address [];

Nel campo `Common Name` specificare un nome per la Certification Authority, per esempio *myserver.com*. Inserendo come valore '.' il campo viene lasciato vuoto. Viene così creata la directory `demoCA/` all'interno della quale sono stati generati i file (*cacert.pem*) e (*cakey.pem*). Il primo contiene la codifica ASCII della codifica binaria *Distinguished Encoding Rules* (*DER*) del certificato della Certification Authority, mentre il secondo contiene la codifica ASCII della codifica binaria

DER della chiave privata RSA a 1024bit. La codifica finale viene chiamata *codifica PEM*.

3.4.2 Creazione della richiesta di certificazione per il server LDAP

Adesso è necessario creare la richiesta di certificazione *Certificate Signing Request* (*CSR*) per il server LDAP. Per fare questo deve essere eseguito il seguente comando:

```
openssl req -newkey rsa:1024 -nodes -keyout newreq.pem -out newreq.pem
```

Durante la procedura è richiesto l'inserimento di un valore per i seguenti campi:

- Country Name (2 letter code) [AU];
- State or Province Name (full name) [Some-State];
- Locality Name (eg, city) [];
- Organization Name (eg, company) [Internet Widgits Pty Ltd];
- Organizational Unit Name (eg, section) [];
- Common Name (eg, YOUR name) [];
- Email Address [];

Nel campo `Common Name` deve essere inserito l'indirizzo del server LDAP (per es. 127.0.0.1), perché questo valore verrà confrontato con l'indirizzo del server al quale il client si connetterà. Verrà inoltre richiesto di inserire una password, che sarà poi inviata insieme alla richiesta di certificazione.

Il risultato di questa operazione è la creazione del file *newreq.pem*. All'interno del file viene inserita la codifica PEM della chiave privata e della richiesta di certificazione.

3.4.3 Creazione del certificato del server da parte della CA

È necessario adesso ottenere dalla Certification Authority il certificato firmato per il server LDAP. Eseguire il seguente comando:

```
CA.sh -sign
```

Alla prima richiesta di password, specificare la password inserita in fase di creazione della Certification Authority. Otterremo così il file (*newcert.pem*). Il file contiene il certificato (contenente la chiave pubblica del server) in formato testo leggibile, la firma della Certification Authority e la codifica PEM del certificato. Nel file *newreq.pem* abbiamo sempre la codifica PEM della chiave privata del server.

Spostiamo il certificato e la chiave privata nella directory precedentemente creata `servercer/`

```
cp demoCA/cacert.pem /home/luca/servercer/cacert.pem
mv newcert.pem /home/luca/servercer/servercrt.pem
mv newreq.pem /home/luca/servercer/serverkey.pem
chmod 400 /home/luca/servercer/serverkey.pem
```

L'ultimo comando rende in sola lettura la chiave privata. Dovrà essere usato il comando *chown* nel caso in cui l'utente che ha generato la chiave privata non sia lo stesso che manda in esecuzione il server di OpenLDAP.

3.4.4 Configurazione del client per SSL

È necessario modificare il file *ldap.conf* nella directory `/usr/local/etc/openldap/`, per specificare l'indirizzo del server LDAP e il path del file contenente la lista dei certificati che il client ritiene validi e fidati:

```
HOST 127.0.0.1
PORT 636

TLS_CACERT /home/luca/servercer/cacert.pem
TLS_REQCERT demand
```

Nel nostro caso basta specificare, per la parola chiave TLS_CACERT, il path del certificato della Certification Authority che ha firmato il certificato del server LDAP (*catena di fiducia*). La variabile TLS_REQCERT specifica quale operazione deve essere compiuta per il certificato del server. I valori possibili sono:

- never: il client non richiede e non controlla il certificato del server;
- allow: il certificato del server è richiesto. Se il certificato del server non viene fornito, la sessione procede ugualmente. Se viene fornito un certificato non valido, viene ignorato e la sessione procede;
- try: Il certificato del server è richiesto. Se il certificato del server non viene fornito, la sessione procede ugualmente. Se viene fornito un certificato non valido, la sessione viene immediatamente terminata;
- demand — hard: queste parole chiavi sono equivalenti. Il certificato del server è richiesto. Se il certificato non viene fornito, o il certificato non è

valido, la sessione viene immediatamente terminata. Questo è il valore di default;

Per quanto riguarda l'EJB, è necessario inserire il certificato del server LDAP nel file contenente la lista dei certificati ritenuti validi e fidati. Questo file si chiama *cacerts* e si trova nella directory
`/home/luca/j2sdk1.4.2_06/jre/lib/security/`.
Per aggiungere il certificato del server al file *cacerts*, si utilizza il tool *keytool* messo a disposizione da Java 2 Standard Edition. Lo strumento keytool accetta i certificati con estensione *crt*, contenenti solamente la codifica PEM, quindi è necessario fare una conversione del certificato del server nella directory `servercer/`:

```
openssl x509 -in servercrt.pem -out server.crt
```

Adesso è possibile aggiungere il certificato al *cacerts*. Dopo essersi spostati nella directory `/home/luca/j2sdk1.4.2_06/bin/` eseguire il comando:

```
./keytool -import -alias servercer -file /home/luca/servercer/server.crt
        -keystore /home/luca/j2sdk1.4.2_06/jre/lib/security/cacerts
```

Con l'opzione *-alias* è possibile specificare un identificatore per il certificato. Verrà richiesta una password, che se non è stata cambiata, risulta essere quella di default *changeit*.

Capitolo 4

EJB LdapServer

4.1 Implementazione dello Stateless Session Bean LdapServer

Per la gestione degli utenti e dei ruoli nel server LDAP, è stato scelto l'utilizzo di un EJB. Il tipo di EJB utilizzato è uno *Stateless Session Bean*. Un EJB di questo tipo non è persistente, cioè i suoi dati non sono salvati su un database. Quando il client invoca il metodo di uno stateless bean, le variabili dell'istanza del bean possono contenere informazioni di stato, ma solo per la durata dell'invocazione. Quando il metodo termina, lo stato non è più mantenuto. In figura **4.1** è mostrato il ciclo di vita dello Stateless Session Bean.

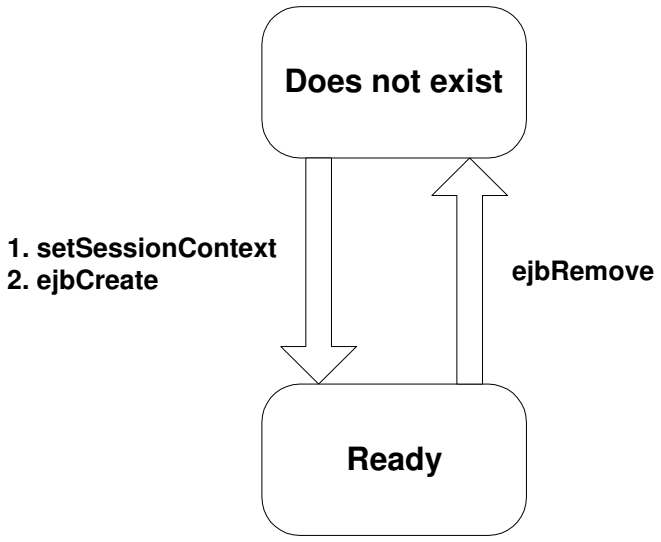

Figura 4.1: Ciclo di vita di uno Stateless Session Bean

Eccetto durante l'invocazione di un metodo, tutte le istanze di uno stateless bean sono equivalenti, permettendo al container EJB di assegnare un'istanza a ciascun client. Dato che gli Stateless Session Bean possono servire più client, essi possono offrire una migliore scalabilità per le applicazioni che richiedono un largo numero di client. A differenza degli *Stateful Session Bean*, lo Stateless Session Bean non scrive mai su memoria di massa, rendendolo così migliore da un punto di vista delle prestazioni. Infine, solo un client alla volta può accedere all'istanza del bean, non può essere condivisa. Conviene utilizzare uno Stateless Session Bean quando:

- lo stato del bean non deve mantenere dati per uno specifico client;
- in una singola invocazione del metodo, il bean compie un generico task per tutti i client;

- il bean recupera dal database un insieme di dati in sola lettura, che sono usati spesso dai client;

Per l'EJB è necessario creare:

- interfaccia *Home*;
- interfaccia *Remote* o *Local*;
- implementazione dell'interfaccia *Remote* o *Local*;
- il deployment descriptor;

Nel nostro caso è stato scelto di implementare il servizio con interfaccia Remote, permettendo così al client di poter essere eseguito su una macchina differente da quella in cui viene eseguito l'EJB. La comunicazione tra i due host viene gestita in modo automatico dall'Application Server sfruttando il supporto di *Java RMI* (*Java Remote Method Invocation*). In questo modo viene garantito il concetto della *trasparenza* per il programmatore del client, in quanto le chiamate dei metodi remoti del Bean, hanno lo stesso tipo di signature delle chiamate di metodi locali.

4.2 Interfaccia Home dell'EJB

L'interfaccia Home estende l'interfaccia *javax.ejb.EJBHome*. In questa interfaccia si definisce il metodo create che un client remoto può invocare:

```
public interface LdapServerHome extends javax.ejb.EJBHome {
 public static final String JNDI_NAME="LdapServer";

 public cmos.LdapServer create()
```

```
  throws javax.ejb.CreateException,java.rmi.RemoteException;
}
```

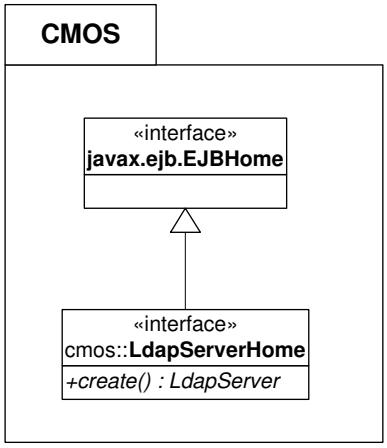

Figura 4.2: Class diagram per la Home Interface

Il metodo `create` corrisponde al metodo `ejbCreate` della classe che implementa l'EJB. Le signature dei due metodi sono simili, ma differiscono per alcuni aspetti importanti. Le regole per definire correttamente il metodo `create` sono:

- il numero e i tipi degli argomenti devono essere gli stessi dell'`ejbCreate`;
- il tipo degli argomenti e del ritorno, devono essere tipi RMI, cioè serializzabili;
- il metodo deve tornare il tipo dell'interfaccia Remote del bean, mentre l'`ejbCreate` torna void;
- l'eccezioni lanciate dal metodo possono essere java.rmi.RemoteException e javax.ejb.CreateException;

Il membro JNDI_NAME specifica il nome con cui è possibile fare il lookup nel servizio JNDI, per ottenere un riferimento remoto all'oggetto Home, col quale

poi invocare il metodo `create`. In realtà il riferimento remoto all'oggetto Home è un riferimento allo stub presente sulla macchina del client che funziona da proxy, perché intercetta le chiamate del client e si preoccupa di passarle allo stub sull'host in cui è presente l'istanza dell'oggetto Home.

4.3 Interfaccia Remote dell'EJB

L'interfaccia Remote estende *javax.ejb.EJBObject*, e definisce i metodi business che un client può invocare. I metodi che si devono definire sono:

```
public interface LdapServer extends javax.ejb.EJBObject {
 public void insertUser(java.util.HashMap valori,java.lang.String schema)
  throws javax.naming.NamingException, java.rmi.RemoteException;

 public void deleteUser(java.lang.String userid,java.lang.String ou) throws ↩
     javax.naming.NamingException, java.rmi.RemoteException;

 public void modifyUser(java.util.HashMap nuoviValori,java.lang.String ↩
     userid,java.lang.String ou) throws javax.naming.NamingException, java. ↩
     rmi.RemoteException;

 public javax.naming.directory.Attributes query(java.lang.String userid,java ↩
     .lang.String ou) throws javax.naming.NamingException, java.rmi. ↩
     RemoteException;

 public java.util.HashSet schemaQuery(java.lang.String schemaNome) throws ↩
     javax.naming.NamingException, java.rmi.RemoteException;

 public void insertRole(java.lang.String ruolo,java.lang.String ou) throws ↩
     javax.naming.NamingException, java.rmi.RemoteException;

 public void deleteRole(java.lang.String ruolo,java.lang.String ou) throws ↩
     javax.naming.NamingException, java.rmi.RemoteException;
```

```
 public void addUserToRole(java.lang.String userid,java.lang.String ouUtente ↩
     ,java.lang.String role,java.lang.String ouRole) throws javax.naming. ↩
     NamingException, java.rmi.RemoteException;

 public void delUserFromRole(java.lang.String userid,java.lang.String role, ↩
     java.lang.String ouRole) throws javax.naming.NamingException, java.rmi. ↩
     RemoteException;
}
```

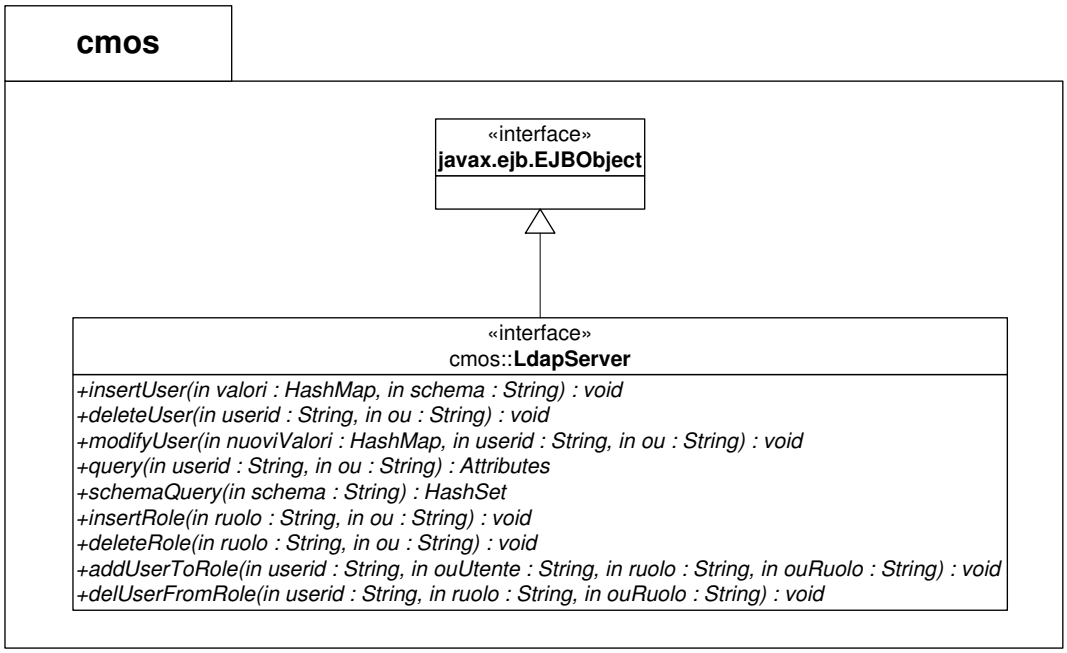

Figura 4.3: Class diagram per Remote Interface

La definizione dei metodi deve seguire le seguenti regole:

- ciascun metodo deve essere implementato nella classe dell'EJB;
- le signature dei metodi devono essere identiche a quelle presenti nella classe dell'EJB;
- i tipi degli argomenti e del ritorno devono essere tipi RMI, cioè serializzabili;

- deve essere lanciata almeno l'eccezione java.rmi.RemoteException;

4.4 Implementazione dell'interfaccia Remote

Per implementare correttamente l'interfaccia Remote, è necessario seguire le seguenti regole:

- deve implementare l'interfaccia *SessionBean*;
- la classe deve essere definita come `public`;
- deve implementare uno o più metodi `ejbCreate`;
- deve implementare i metodi business definiti nell'interfaccia Remote;

I metodi business permettono di compiere le seguenti operazioni sul server LDAP:

- inserimento di un nuovo utente (`insertUser`);
- cancellazione di un utente esistente (`deleteUser`);
- modifica dei valori degli attributi di un utente (`modifyUser`);
- ricavare i valori degli attributi di un utente (`query`);
- ricavare gli attributi di uno schema (`schemaQuery`);
- inserimento di un nuovo ruolo (`insertRole`);
- cancellazione di un ruolo esistente (`deleteRole`);
- aggiungere un utente ad uno specifico ruolo (`addUserToRole`);

- cancellazione di un utente da uno specifico ruolo (`delUserFromRole`);

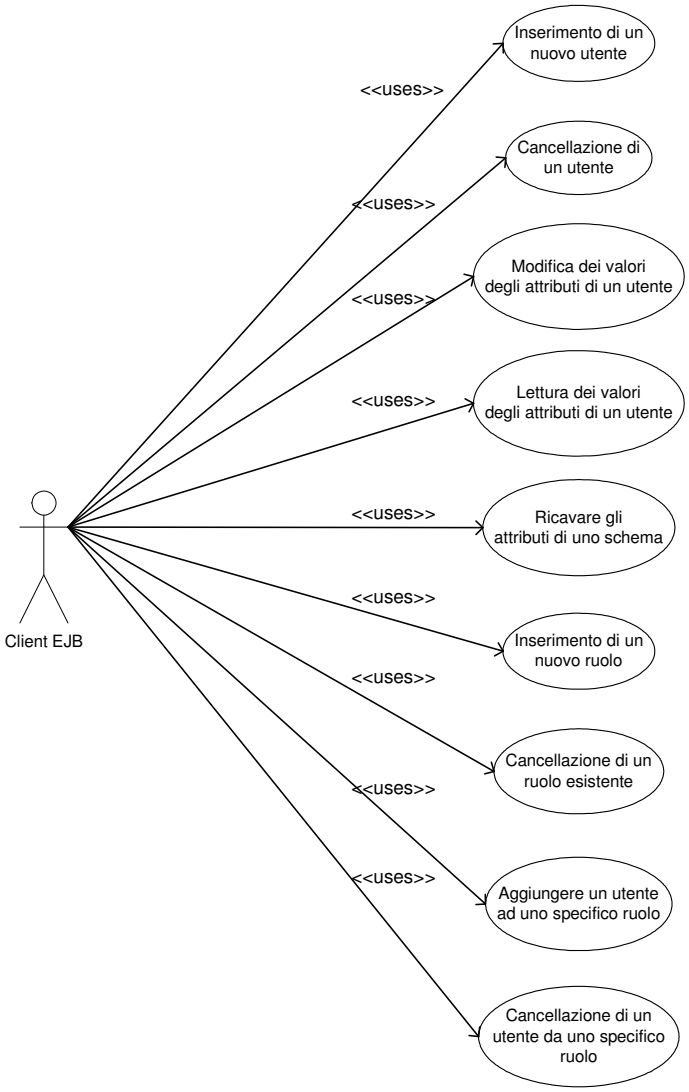

Figura 4.4: Use case diagram dell'EJB LdapServer

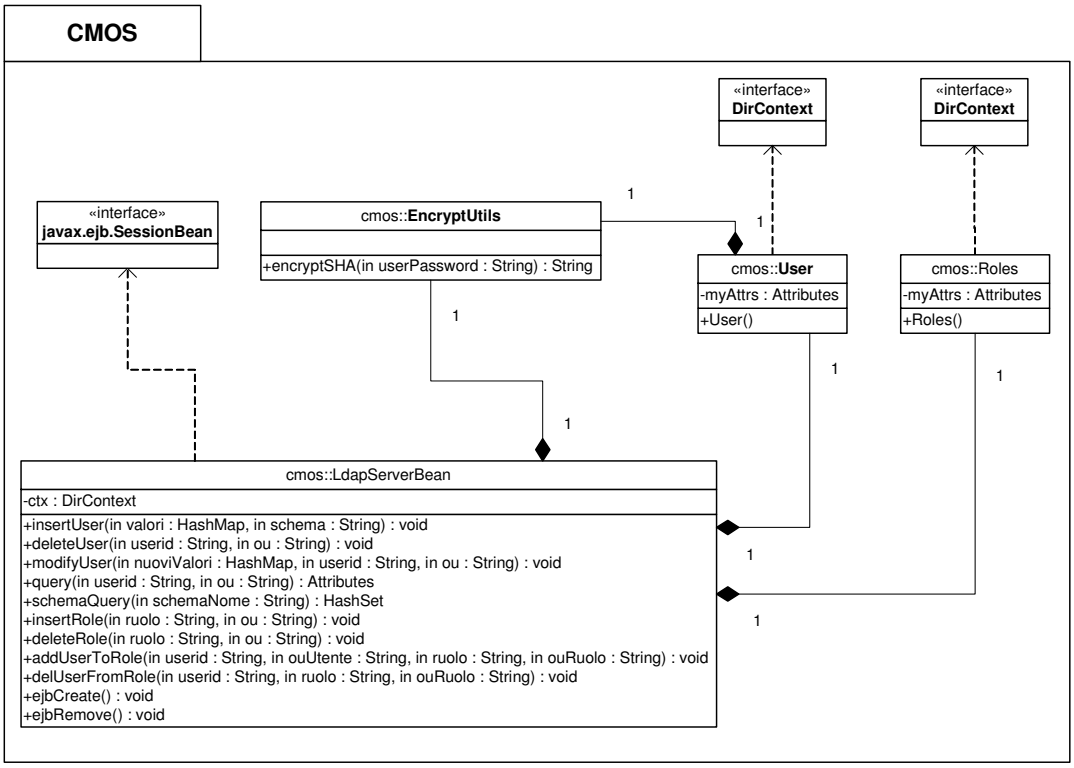

Figura 4.5: Class diagram per la classe LdapServerBean

Per prima cosa è necessario inizializzare un nuovo contesto di tipo *InitialDirContext*. Questa operazione viene eseguita nell'`ejbCreate`, in modo tale che il container possa generare delle istanze identiche dell'EJB, tutte caratterizzate dallo stesso contesto. Per inizializzare il contesto specifichiamo i valori delle variabili d'ambiente per il server LDAP:

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,"com.sun.jndi.ldap.LdapCtxFactory");
env.put(Context.PROVIDER_URL, "ldaps://127.0.0.1/o=myserver.com");
env.put(Context.SECURITY_AUTHENTICATION, "simple");
env.put(Context.SECURITY_PRINCIPAL,"cn=Manager,o=myserver.com");
env.put(Context.SECURITY_CREDENTIALS,"openldap");
try {
 ctx = new InitialDirContext(env);
```

```
} catch (NamingException ne) {
 System.err.println("[ERRORE] NamingException: "+ne);
}
```

Le variabili d'ambiente per il server LDAP ci permettono di specificare:

- l'url del server compreso il Distinguished Name del contesto root. In questo modo tutti i Distinguished Name delle entry che vengono inviati al server LDAP, non devono comprendere la parte relativa al contesto root, in quanto già specificata nella variabile d'ambiente;
- abilitare il canale SSL tra l'EJB e il server LDAP, utilizzando la parola chiave *ldaps*;
- tipo di autenticazione del client richiesta, in questo caso è stato scelto *simple* **§3.1.1**;
- Distinguished Name e password del rootdn specificati nel file di configurazione di OpenLDAP *slapd.conf* **§3.2**;

Ciascuna istanza del Pool stabilisce una connessione SSL con il server LDAP che mantiene attiva finché il container non decide di invocare il metodo `ejbRemove`, all'interno del quale viene chiuso il contesto:

```
ctx.close();
```

I metodi business sono stati implementati con riferimento al Directory Information Tree ad un solo livello **§5.1**.

4.4.1 Inserimento di un nuovo utente

Il metodo `insertUser` permette di inserire un nuovo utente all'interno del server LDAP. I parametri di ingresso sono i valori degli attributi e il nome dello schema che individua l'objectClass. Viene creato un oggetto di tipo *User*, passando come parametri del costruttore, i valori degli attributi e il nome dello schema. La classe User implementa l'interfaccia *DirContext* e presenta come membro un oggetto di tipo *Attributes*. L'interfaccia Attributes rappresenta una collezione di attributi, ed è possibile aggiungerli utilizzando il metodo `put`, specificando come parametri il nome e il valore dell'attributo. È necessario inserire tra la lista degli attributi, anche gli objectClass, che nel caso dell'utente, assumono i valori *top*, *person*, *organizationalPerson*, *inetOrgPerson* e il nome dello schema passato per parametro al costruttore. Il valore dell'attributo `userPassword`, prima di essere inserito nell'oggetto Attributes, viene passato a una funzione che implementa la funzione Hash SHA, concatenando al risultato il prefisso SHA, per permettere al LdapLoginModule **§3.1.1** di riconoscere la funzione Hash utilizzata.

Al fine di implementare l'interfaccia DirContext, la classe User deve definire una serie di metodi che non verranno poi utilizzati.

Dopo aver creato l'oggetto di tipo User, il metodo `insertUser` utilizza il metodo `bind` del DirContext *ctx* per creare l'entry nel server LDAP, e compiere l'operazione di naming, associando all'oggetto User il Distinguished Name esclusa la radice, in quanto già specificata nell'inizializzazione del contesto nella variabile d'ambiente context.PROVIDER_URL:

```
User p = new User(valori,schema);
```

```
ctx.bind("uid="+(String)valori.get("uid")+",ou="+(String)valori.get("ou"), p ↩
    );
```

Il Distinguished Name è caratterizzato dall'attributo `uid` (userid) dell'objectClass inetOrgPerson, e dall'attributo `ou` (organizationalUnit) che deve assumere come valore uno dei subContext creati precedentemente. Nel nostro caso dovrà assumere come valore l'organizationalUnit *People*.

Il metodo bind può generare le seguenti eccezioni:

- NameAlreadyBoundException se esiste già un oggetto con il Distinguished Name specificato;
- InvalidAttributesException se qualche attributo obbligatorio non è stato inserito nell'oggetto Attributes;
- NamingException se si verifica un eccezione di naming;

4.4.2 Cancellazione di un utente esistente

Il metodo `deleteUser` permette di rimuovere un'entry relativa ad un utente nel server LDAP. I parametri di ingresso sono il valore dell'attributo `uid` e `ou` dell'utente, in modo da individuare l'entry in modo univoco attraverso il Distinguished Name. Questi valori vengono forniti al metodo `verify` per verificare l'esistenza dell'entry. Il metodo `verify` prende in ingresso il Distinguished Name e utlizza il metodo `getAttributes` del DirContext *ctx* per ricavare gli attributi dell'entry. Se specifichiamo nel metodo `getAttributes` come lista degli attributi un Array di stringhe vuoto, non viene recuperato nessun valore degli attributi. Se l'entry non esiste viene lanciata un eccezione di tipo NamingException altrimenti torna l'intero 1.

```
String[] attrID = {};
attributi = ctx.getAttributes(DN,attrID);
```

Se il metodo `verify` ha successo, viene poi chiamato all'interno della funzione `deleteUser`, il metodo `unbind` del DirContext ctx, specificando il Distinguished Name dell'entry utente che vogliamo rimuovere dal server LDAP:

```
if(verify("uid=" + userid + ",ou=" + ou) == 1)
 ctx.unbind("uid=" + userid + ",ou=" + ou);
```

Il metodo `unbind` può generare le seguenti eccezioni:

- NameNotFoundException se un contesto intermedio non esiste;
- NamingException se si verifica un'eccezione di naming;

Come è possibile vedere, il metodo `unbind` non genera un'eccezione nel caso in cui il Relative Distinguished Name dell'entry che vogliamo rimuovere non sia valido. Per questo è stato necessario implementare ed utilizzare il metodo `verify`, per essere sicuri che la cancellazione non abbia successo nel caso in cui l'entry non esista.

4.4.3 Modifica dei valori degli attributi

Il metodo `modifyUser` permette di modificare i valori degli attributi di un utente all'interno del server LDAP. I parametri di ingresso sono un *HashMap* contenente le coppie (nome,nuovo valore) degli attributi, e il valore degli attributi `uid` e `ou` per la creazione del Distinguished Name dell'entry. Per ciascun elemento dell'HashMap viene creato un oggetto *ModificationItem*, il cui costruttore richiede il nome della modifica da fare, e l'attributo costituito dal nome e dal nuovo valore. Il nome della modifica può assumere i seguenti valori:

- DirContext.ADD_ATTRIBUTE per aggiungere un nuovo attributo;
- DirContext.REPLACE_ATTRIBUTE per modificare il valore dell'attributo;
- DirContext.REMOVE_ATTRIBUTE per rimuovere l'attributo;

Nel nostro caso utilizziamo il valore

DirContext.REPLACE_ATTRIBUTE:

```
int mapSize = nuoviValori.size();
ModificationItem[] mods = new ModificationItem[mapSize];
mods[i] = new ModificationItem(DirContext.REPLACE_ATTRIBUTE,
        new BasicAttribute((String)entry.getKey(), (String)entry.getValue()) ↩
            );
```

Per il nuovo valore dell'attributo `userPassword` viene sempre utilizzato il metodo che implementa la funzione Hash, concatenando il prefisso SHA.
Infine il metodo `modifyUser` utilizza il metodo `modifyAttributes` del DirContext *ctx* per aggiornare i valori degli attributi dell'entry utente, specificando come parametri il Distinguished Name dell'entry e l'Array di ModificationItem:

```
ctx.modifyAttributes("uid="+userid+",ou="+ou, mods);
```

Il metodo `modifyAttributes` può generare le seguenti eccezioni:

- AttributeModificationException se le modifiche non possono essere completate con successo;
- NamingException se si verifica un'eccezione di naming;

4.4.4 Recuperare i valori degli attributi

Il metodo `query` permette di ricavare i valori degli attributi di un'entry utente all'interno del server LDAP. I parametri di ingresso sono i valori degli attributi `uid` e `ou` per la costruzione del Distinguished Name. Viene utilizzato il metodo `getAttributes` del DirContext *ctx* specificando come parametro il Distinguished Name, e ritorna un oggetto di tipo *Attributes* contenente la lista degli attributi con i relativi valori:

```
attributi = ctx.getAttributes("uid="+userid+",ou="+ou);
```

Dall'oggetto attributi viene poi rimosso l'attributo *objectClass* in quanto contiene informazioni non utili per il client.

```
attributi.remove("objectClass");
```

Il metodo `getAttributes` può generare le seguenti eccezioni:

- NamingException se si verifica un'eccezione di naming;

4.4.5 Recupero degli attributi dello schema

Il metodo `schemaQuery` permette di recuperare gli attributi MUST e MAY dello schema utilizzato per l'utente, nel nostro caso *musica*. Il parametro di ingresso è il nome dello schema e ritorna un *HashSet* contenente i nome degli attributi. Viene utilizzato il metodo `getSchema` del DirContext *ctx* per recuperare il DirContext associato con gli schemi esistenti nel server LDAP. Da questo nuovo DirContext invochiamo il metodo `lookup` per recuperare il DirContext associato all'objectClass *musica*:

```
DirContext schema = ctx.getSchema("");
```

```
DirContext personSchema = (DirContext)schema.lookup("ClassDefinition/"+ ↩
    schemaNome);
```

Tutti gli objectClass vengono associati al loro nome concatenando il prefisso *ClassDefinition*. Infine utilizziamo il metodo `getAttributes` sul DirContext *personSchema* per recuperare i valori degli attributi MUST e MAY. I valori vengono poi inseriti in un HashSet e ritornati alla funzione chiamante.

4.4.6 Creazione di un ruolo

Il metodo `insertRole` permette di creare un nuovo ruolo all'interno del server LDAP. I parametri di ingresso sono il nome del nuovo ruolo e il valore dell'attributo `ou` che specifica l'organizationalUnit sotto la quale il ruolo deve essere creato. Viene creato un oggetto di tipo *Roles* passando al costruttore il nome del ruolo. La classe Roles implementa l'interfaccia *DirContext* e presenta come membro un oggetto di tipo *Attributes*. L'interfaccia Attributes rappresenta una collezione di attributi, ed è possibile aggiungerli utilizzando il metodo `put`, specificando come parametri il nome e il valore degli attributi. È necessario inserire tra la lista degli attributi, anche gli objectClass, che nel caso del ruolo, assumono i valori *top* e *organizationalRole*. Nel nostro caso si aggiungono gli attributi `cn` (commonName) obbligatorio per l'objectClass organizationalRole, e l'attributo `l` (localityName) per il nome del ruolo. Al fine di implementare l'interfaccia DirContext, la classe Roles deve definire una serie di metodi che non verranno poi utilizzati.

Dopo aver creato l'oggetto di tipo Roles, il metodo `insertRole` utilizza il metodo `bind` del DirContext *ctx* per creare l'entry nel server LDAP, e compie-

re l'operazione di naming, associando all'oggetto Roles il Distinguished Name esclusa la radice, in quanto già specificata nell'inizializzazione del contesto nella variabile d'ambiente context.PROVIDER_URL:

```
Roles r = new Roles(ruolo);
ctx.bind("cn="+ruolo+",ou="+ou, r);
```

Il Distinguished Name è caratterizzato dall'attributo `cn` (commonName) dell'objectClass organizationalRole, e dall'attributo `ou` (organizationalUnit) che deve assumere come valore uno dei subContext creati precedentemente. Nel nostro caso dovrà assumere come valore l'organizationalUnit *Roles*.
Il metodo `bind` può generare le seguenti eccezioni:

- NameAlreadyBoundException se esiste già un oggetto con il Distinguished Name specificato;
- InvalidAttributesException se qualche attributo obbligatorio non è stato inserito nell'oggetto Attributes;
- NamingException se si verifica un eccezione di naming;

4.4.7 Cancellazione di un ruolo

Il metodo `deleteRole` permette di rimuovere un'entry relativa ad un ruolo nel server LDAP. I parametri di ingresso sono il nome del ruolo e il valore dell'attributo `ou`, in modo da individuare l'entry in modo univoco attraverso il Distinguished Name. Questi valori vengono forniti al metodo `verify` per verificare l'esistenza dell'entry. Il metodo `verify` prende in ingresso il Distinguished Name e utlizza il metodo `getAttributes` del DirContext *ctx* per ricavare gli

attributi dell'entry. Se specifichiamo nel metodo `getAttributes` come lista degli attributi un Array di stringhe vuoto, non viene recuperato nessun valore degli attributi. Se l'entry non esiste viene lanciata un eccezione di tipo NamingException altrimenti torna l'intero 1.

```
String[] attrID = {};
attributi = ctx.getAttributes(DN,attrID);
```

Se il metodo `verify` ha successo, viene poi chiamato all'interno della funzione `deleteRole`, il metodo `unbind` del DirContext ctx, specificando il Distinguished Name dell'entry ruolo che vogliamo rimuovere dal server LDAP:

```
if(verify("cn="+ruolo+",ou="+ou) == 1)
 ctx.unbind("cn="+ruolo+",ou="+ou);
```

Il metodo `unbind` può generare le seguenti eccezioni:

- NameNotFoundException se un contesto intermedio non esiste;
- NamingException se si verifica un'eccezione di naming;

Come è possibile vedere, il metodo `unbind` non genera un'eccezione nel caso in cui il Relative Distinguished Name dell'entry che vogliamo rimuovere non sia valido. Per questo è stato necessario implementare ed utilizzare il metodo `verify`, per essere sicuri che la cancellazione non abbia successo nel caso in cui l'entry non esista.

4.4.8 Aggiungere un utente ad un ruolo

Il metodo `addUserToRole` permette di aggiungere un utente esistente ad un ruolo esistente nel server LDAP. I parametri di ingresso sono i valori degli attributi `uid` e `ou` per costruire il Distinguished Name relativo all'utente, e il

nome del ruolo e il suo organizationalUnit per costruire il Distinguished Name relativo al ruolo. Viene utilizzata la funzione `verify` per verificare l'esistenza dell'entry utente, e in caso di successo, viene creato un oggetto di tipo *ModificationItem* passando al costruttore l'operazione DirContext.ADD_ATTRIBUTE, e il nuovo attributo `description` che assume come valore l'userid dell'utente. Questo attributo `description` deve essere aggiunto all'entry relativa al ruolo specificato. Per fare questo viene utilizzato il metodo `modifyAttributes` del DirContext *ctx*, passando come parametri il Distinguished Name del ruolo e l'oggetto ModificationItem creato precedentemente:

```
if(verify("uid="+userid+",ou="+ouUtente) == 1) {
 mods = new ModificationItem[1];
 mods[0] = new ModificationItem(DirContext.ADD_ATTRIBUTE,new BasicAttribute( ↩
     "description", userid));
 try {
  ctx.modifyAttributes("cn="+role+",ou="+ouRole, mods);
 } catch (NamingException ne) {
  throw new NamingException("[ERRORE] NamingException: "+ne);
 }
}
```

Il metodo `modifyAttributes` può generare le seguenti eccezioni:

- AttributeModificationException se le modifiche non possono essere completate con successo;
- NamingException se si verifica un'eccezione di naming;

4.4.9 Cancellazione di un utente da un ruolo

Il metodo `delUserFromRole` permette di cancellare un utente da un ruolo all'interno del server LDAP. I parametri di ingresso sono il valore dell'attributo `uid`

dell'utente, il nome del ruolo e l'organizatrionalUnit del ruolo necessari per la costruzione del Distinguished Name. Viene creato un oggetto di tipo *ModificationItem* passando al costruttore l'operazione DirContext.REMOVE_ATTRIBUTE e l'attributo `description` con valore l'userid dell'utente. Per rimuovere l'utente dal ruolo basta quindi rimuovere dall'entry del ruolo, l'attributo `description` che assume come valore l'userid dell'utente. Per fare questo si utilizza il metodo `modifyAttributes` del DirContext *ctx*, specificando come attributi il Distinguished Name del ruolo e l'oggetto ModificationItem appena creato:

```
ModificationItem[] mods = new ModificationItem[1];
try {
 mods[0] = new ModificationItem(DirContext.REMOVE_ATTRIBUTE,new ↩
     BasicAttribute("description",userid));
 ctx.modifyAttributes("cn="+role+",ou="+ouRole, mods);
} catch (NamingException ne) {
 throw new NamingException("[ERRORE] NamingException: "+ne);
}
```

Il metodo modifyAttributes può generare le seguenti eccezioni:

- AttributeModificationException se le modifiche non possono essere completate con successo;
- NamingException se si verifica un'eccezione di naming;

4.5 Deployment descriptor

Il *deployment descriptor* deve essere chiamato *ejb-jar.xml* e deve essere creato nella directory `META-INF/`. Questo file dice al server EJB, quali classi implementano il bean, l'interfaccia Home e Remote. Se nel package c'è più di un

EJB, il deployment descriptor indica come gli EJB interagiscano tra di loro. Sotto viene riportato il contenuto del file *ejb-jar.xml* per l'esempio:

```
<ejb-jar >
 <enterprise-beans>
  <session>
   <ejb-name>LdapServer</ejb-name>
   <home>cmos.LdapServerHome</home>
   <remote>cmos.LdapServer</remote>
   <ejb-class>cmos.LdapServerSession</ejb-class>
   <session-type>Stateless</session-type>
   <transaction-type>Container</transaction-type>
  </session>
 </enterprise-beans>

 <assembly-descriptor>
  <security-role>
   <role-name>Administrator</role-name>
  </security-role>
  <security-role>
   <role-name>User</role-name>
  </security-role>

  <method-permission>
   <role-name>Administrator</role-name>
   <method>
    <ejb-name>LdapServer</ejb-name>
    <method-name>*</method-name>
   </method>
  </method-permission>
  <method-permission>
   <role-name>User</role-name>
   <method>
    <ejb-name>LdapServer</ejb-name>
    <method-name>query</method-name>
   </method>
   <method>
```

```
    <ejb-name>LdapServer</ejb-name>
    <method-name>create</method-name>
   </method>
  </method-permission>
 </assembly-descriptor>
</ejb-jar>
```

Le seguenti sezioni permettono di specificare:

- <ejb-name>: nome dell'EJB;
- <home>: nome dell'interfaccia Home;
- <remote>: nome dell'interfaccia Remote;
- <ejb-class>: nome della classe che implementa l'EJB;
- <session-type>: tipo di EJB, nel nostro caso Stateless Session Bean;
- <transaction-type>: tipo di transaction management;
- <role-name>: nome del ruolo con il quale l'utente può accedere all'EJB. Usando il LoginModule *LdapLoginModule*, i nomi dei ruoli devono essere uguali al valore che l'attributo `l` assume nell'entry del ruolo all'interno del server LDAP;
- <method-permission>: in questa sezione si specifica il ruolo che l'utente deve avere per accedere a un particolare insieme di metodi;
- <method-name>: nome del metodo al quale l'utente, con il ruolo specificato precedentemente, può accedere;

Per ciascun ruolo si deve specificare l'insieme dei metodi ai quali è permesso accedere. Con il valore '*' si specifica che l'utente può accedere a tutti i metodi dell'EJB. Nell'esempio è necessario creare anche un altro deployment descriptor, per modificare il comportamento di default di JBoss. Il file deve essere chiamato *jboss.xml*, e deve essere creato sempre nella directory `META-INF/`. Sotto è riportato il contenuto del file:

```
<jboss>
 <security-domain>java:/jaas/ldap</security-domain>
 <enterprise-beans>
  <session>
   <ejb-name>LdapServer</ejb-name>
   <jndi-name>LdapServer</jndi-name>
  </session>
 </enterprise-beans>
</jboss>
```

Le seguenti sezioni permettono di specificare:

- <security-domain>: nome della Configurazione specificata per il LoginModule *LdapLoginModule*, all'interno del file *auth.conf*;
- <ejb-name>: nome dell'EJB specificato nel file *ejb-jar.xml*;
- <jndi-name>: nome JNDI con cui il client può recuperare il riferimento remoto all'oggetto Home. Può essere diverso dal valore specificato nel tag <*ejb-name*>, ma deve essere lo stesso valore specificato nell'interfaccia Home dell'EJB **§4.2**;

Capitolo 5

Gestione dei contesti nel server LDAP

5.1 Implementazione del client stand-alone LdapClientContext

È necessario implementare un tool per la gestione dei contesti nel server LDAP.

Il file è stato chiamato *LdapClientContext.java* e fornisce dei metodi per:

- creazione del contesto root;
- creazione di un subContext;
- cancellazione di un subContext esistente;

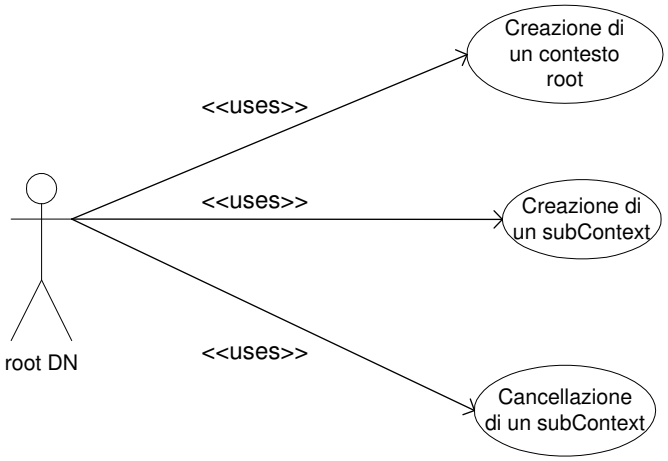

Figura 5.1: Use case diagram per il client LdapClientContext

Il tool permette di generare un Directory Information Tree, in cui la root è un'entry che presenta come attributo l'objectClass `organization`, e i nodi intermedi che rappresentano i subContext, sono entry che presentano come attributo l'objectClass `organizationalUnit`. Si deve precisare che il Distinguished Name del contesto root che si deve creare, è lo stesso presente nel file di configurazione di OpenLDAP (*slapd.conf*) alla direttiva *suffix*, nel nostro caso *o=myserver.com*. La scelta dell'attributo per il prefisso del Distinguished Name del contesto root, può avvenire tra uno degli attributi messi a disposizione dall'objectClass `organization`.

Il Distinguished Name del subContext deve presentare come prefisso, uno dei possibili attributi messi a disposizione dall'objectClass `organizationalUnit`, e come suffisso il Distinguished Name del nodo superiore, che può essere il rootContext ma anche un altro subContext. Nel nostro caso i subContext creati sono tre:

- People: per memorizzare entry relative agli utenti;

- Roles: per memorizzare entry relative ai ruoli per l'accesso all'EJB;
- BeanAccess: per memorizzare entry relative agli utenti che possono accedere all'EJB;

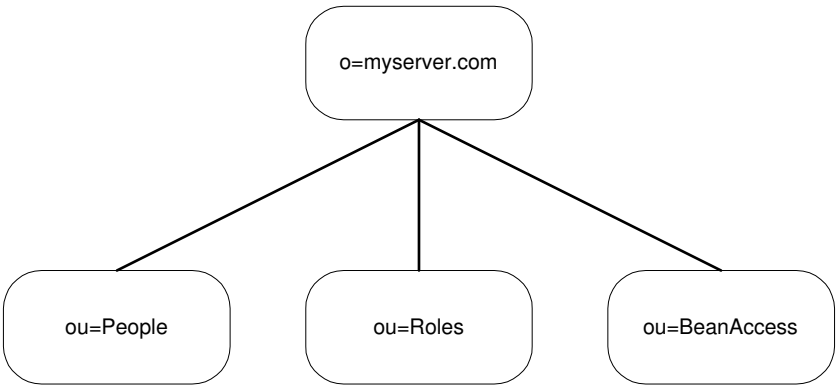

Figura 5.2: Directory Information Tree per l'esempio

La classe LdapClientContext presenta i seguenti metodi:

```
public void contestoRoot();
public void sottoContesto();
public void cancellaSottoContesto();
private int verify(String subContext) throws NamingException;
private void principale();
public static void main(String[] args);
```

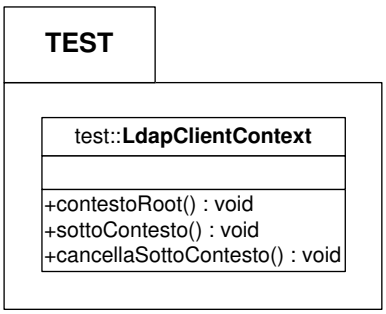

Figura 5.3: Class diagram per la classe LdapClientContext

Nel metodo `principale` avvengono le chiamate ai metodi che compiono le operazioni principali per la gestione dei contesti. Prima di mettere a disposizione dell'utente queste operazioni, è necessario configurare un contesto di directory *DirContext*. All'utente sarà richiesto di specificare da console l'indirizzo del server LDAP, il rootdn e la passwd inseriti nel file di configurazione *slapd.conf* **§3.2**. Il contesto viene così inizializzato:

```
env.put(Context.INITIAL_CONTEXT_FACTORY,"com.sun.jndi.ldap.LdapCtxFactory");
System.out.print("Inserire l'indirizzo del server LDAP: ");
ldapServer = br.readLine();
ldapServer.trim();
env.put(Context.PROVIDER_URL, "ldaps://"+ldapServer+"/");
env.put(Context.SECURITY_AUTHENTICATION, "simple");
System.out.print("Inserire root DN: ");
rootDN = br.readLine();
rootDN.trim();
env.put(Context.SECURITY_PRINCIPAL,rootDN);
System.out.print("Inserire root password: ");
password = br.readLine();
password.trim();
env.put(Context.SECURITY_CREDENTIALS,password);
ctx = new InitialDirContext( env );
```

env è un HashTable e ctx è un oggetto privato di tipo DirContext.

5.1.1 Tool come client stand-alone

Il tool è stato implementato come client stand-alone che stabilisce un collegamento SSL direttamente con il server LDAP, evitando di usare possibili metodi remoti dell'EJB. Questo perché all'interno del server LDAP non sono ancora presenti le entry relative al controllo degli accessi all'EJB e quindi l'utilizzo di quest'ultimo non sarebbe funzionante da un punto di vista della sicurezza.

È possibile comunque utilizzare questo tool, sempre per la gestione dei contesti nel server LDAP, anche dopo che l'EJB funziona correttamente da un punto di vista del controllo degli accessi.

5.1.2 Creazione di un rootContext

Il metodo `contestoRoot()` permette di creare il contesto root. Quando viene eseguito, viene richiesto all'utente di specificare sulla console il Distinguished Name del contesto root da creare. Il valore viene poi passato come parametro alla funzione `createSubcontext` del DirContext ctx, insieme agli objectClass `organization` e `top`:

```
ctx.createSubcontext(rootContext,objectClass);
```

Nel caso in cui il Distinguished Name non sia valido, viene generata un'eccezione di tipo *NamingException* e l'operazione fallisce.

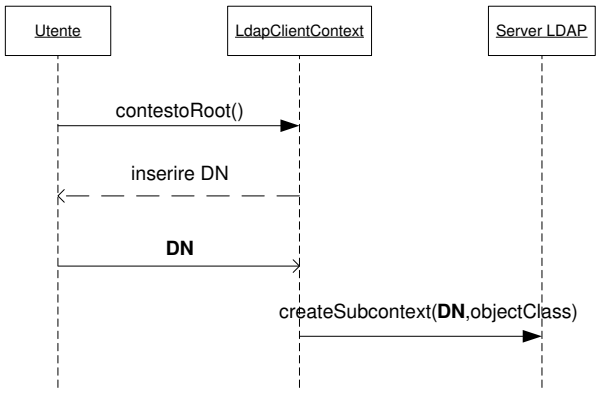

Figura 5.4: Sequence diagram per la creazione di un contesto root

5.1.3 Creazione di un subContext

Il metodo `sottoContesto()` permette di creare un subContext a partire dal rootContext. Quando viene eseguito, viene richiesto all'utente di specificare su console il Distinguished Name del subContext da creare. Il valore viene poi passato come parametro alla funzione `createSubcontext` del DirContext ctx, insieme agli objectClass `organizationalUnit` e `top`:

```
ctx.createSubcontext(subContext,objectClass);
```

Nel caso in cui il Distinguished Name non sia valido, viene generata un'eccezione di tipo *NamingException* e l'operazione fallisce.

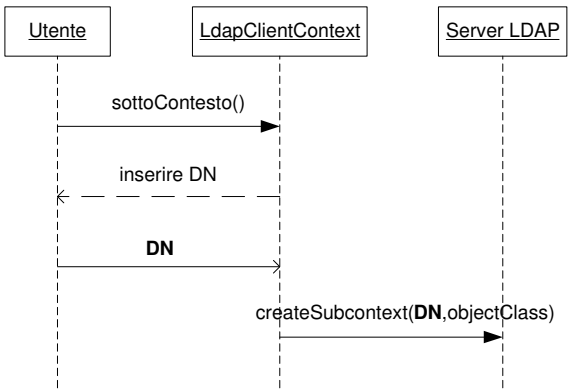

Figura 5.5: Sequence diagram per la creazione di un subContext

5.1.4 Cancellazione di un subContext

Il metodo `cancellaSottoContesto()` permette di cancellare un subContext esistente. Quando viene eseguito, viene richiesto all'utente di specificare su console il Distinguished Name del subContext da cancellare. Il valore viene passato

al metodo `verify` per verificare l'esistenza dell'entry. Il metodo `verify` utilizza la funzione `getAttributes` del DirContext ctx, specificando come parametri il Distinguished Name del quale si vuole ricavare i valori degli attributi (nel nostro caso quello del subContext), e un array di stringhe vuoto:

```
String[] attrID = {};
Attributes attributi = ctx.getAttributes(subContext,attrID);
```

L'array vuoto fa si che non siano ritornati i valori degli attributi, in modo da non rendere pesante l'operazione. Se la funzione `getAttributes` ha successo, significa che l'entry esiste e il metodo `verify` torna 1. Altrimenti la funzione `getAttributes` genera un'eccezione di tipo NaminException che viene catturata dal metodo `verify` e propagata al metodo chiamante `cancellaSottoContesto`. Se il metodo `verify` ha successo, il Distinguished Name viene passato come parametro alla funzione `destroySubcontext` del DirContext ctx:

```
ctx.destroySubcontext(subContext);
```

Contesti intermedi non verranno cancellati, ma solo quello identificato dal Distinguished Name. È da notare che la funzione
`destroySubcontext` ha successo anche se il Relative Distinguished Name del subContext non è valido, mentre può generare un'eccezione di tipo NameNotFoundException nel caso in cui non esiste un contesto intermedio specificato nel Distinguished Name. Per questo è necessario implementare ed utilizzare il metodo `verify`, perché l'utente potrebbe non accorgersi di un errore di battitura del Relative Distinguished Name ed essere convinto che l'operazione sia avvenuta con successo, non ricevendo nessun messaggio di errore.
Inoltre per cancellare un subContext è necessario che sia vuoto, cioè che non siano presenti delle entry.

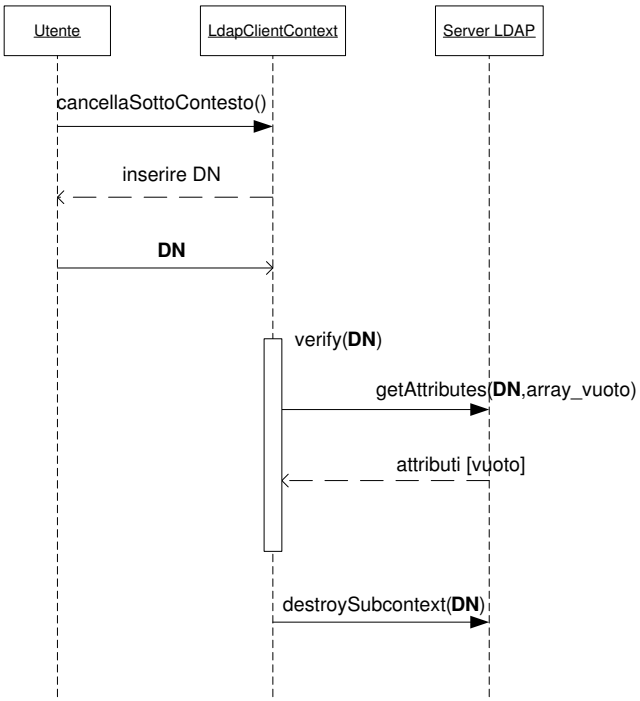

Figura 5.6: Sequence diagram per la cancellazione di un subContext

5.2 Tool ldapadd

OpenLDAP mette a disposizione il tool *ldapadd* per l'inserimento di entry nel server LDAP. È necessario utilizzare questo tool per compiere le seguenti operazioni:

- creazione dei ruoli *Administrator* e *User* sotto il subContext Roles;
- creazione degli utenti `amministratore` e `utente` sotto il subContext BeanAccess;

- inserimento degli utenti `amministratore` e `utente` nei ruoli rispettivamente di Administrator e User;

L'utente `amministratore` ha la possibilità di accedere all'EJB e di utilizzare tutti i metodi remoti messi a disposizione, mentre l'utente `utente` può utilizzare solo i metodi a lui permessi, e specificati nel *deployment descrptor* dell'EJB **§4.5**.

Si deve creare il file *insertUser.ldif* nel formato *ldif*, dove verranno inserite le entry, e dato poi in ingresso al tool ldapadd:

```
# Administrator, Roles, myserver.com
dn: cn=Administrator,ou=Roles,o=myserver.com
l: Administrator
ou: Roles
objectClass: organizationalRole
objectClass: top
cn: Utenti Administrator
description: amministratore

# User, Roles, myserver.com
dn: cn=User,ou=Roles,o=myserver.com
l: User
ou: Roles
objectClass: organizationalRole
objectClass: top
cn: Utenti User
description: utente

# utente, BeanAccess, myserver.com
dn: uid=utente,ou=BeanAccess,o=myserver.com
sn: -
userPassword:: e1NIQX1zOGlwdGFqS0Y5Q3lxT2hVREJsdklRRUtKdUU9
objectClass: inetOrgPerson
objectClass: organizationalPerson
objectClass: person
```

```
objectClass: top
uid: utente
cn: -

# amministratore, BeanAccess, myserver.com
dn: uid=amministratore,ou=BeanAccess,o=myserver.com
sn: -
userPassword:: e1NIQX0wQkwyZ1VUdER4SWRQTU13b1g3c1Vvd3VmVms9
objectClass: inetOrgPerson
objectClass: organizationalPerson
objectClass: person
objectClass: top
uid: amministratore
cn: -
```

In ciascuna entry si deve specificare il Distinguished Name (dn) con cui l'entry deve essere legata. I ruoli sono caratterizzati dall'objectClass `organizationalRole` il quale richiede obbligatorio l'attributo `cn`. Con l'attributo `l` specifichiamo il nome del ruolo, e con l'attributo `description` lo userid degli utenti mappati con lo specifico ruolo. È stato scelto l'attributo `description` perché permette di assumere più valori, nel nostro caso, più utenti mappati nello stesso ruolo.

Gli utenti sono caratterizzati dagli objectClass `inetOrgPerson`, `organizationalPerson` e `person`. Gli attributi obbligatori sono `cn` e `sn`, che nel nostro caso assumono come valore '-' in quanto non utilizzati.

L'attributo `objectClass` è obbligatorio, secondo quanto specificato dall'objectClass `top`.

Nel nostro caso l'userid *amministratore*, viene anche inserito come valore dell'attributo `description` del ruolo *Administrator*, mentre l'userid *utente* viene inserito nell'attributo `description` del ruolo *User*.

Per inserire queste entry nel server LDAP, utilizzare il seguente comando come utente root:

```
ldapadd -f /home/luca/inserUser.ldif -x -D "cn=Manager,o=myserver.com"
        -H ldaps://127.0.0.1 -W
```

Con le opzioni si specifica:

- -f il path del file contenente le entry;
- -x autenticazione del client di tipo *simple* **§3.1.1**;
- -D root DN;
- -H indirizzo del server LDAP, e con il suffisso 's' l'utilizzo di SSL;
- -W richiesta della password rootpw per l'autenticazione del client;

Capitolo 6

Gestione degli utenti nel server LDAP

6.1 Implementazione del client LdapClientUsers

La classe *LdapClientUsers.java* permette di gestire gli utenti nel server LDAP, offrendo dei metodi che interagiscono con l'EJB *LdapServer*. Le operazioni messe a disposizione dal client sono:

- lettura dei valori degli attributi dell'utente;
- modifica dei valori degli attributi dell'utente;
- inserimento di un nuovo utente;
- cancellazione di un utente esistente;

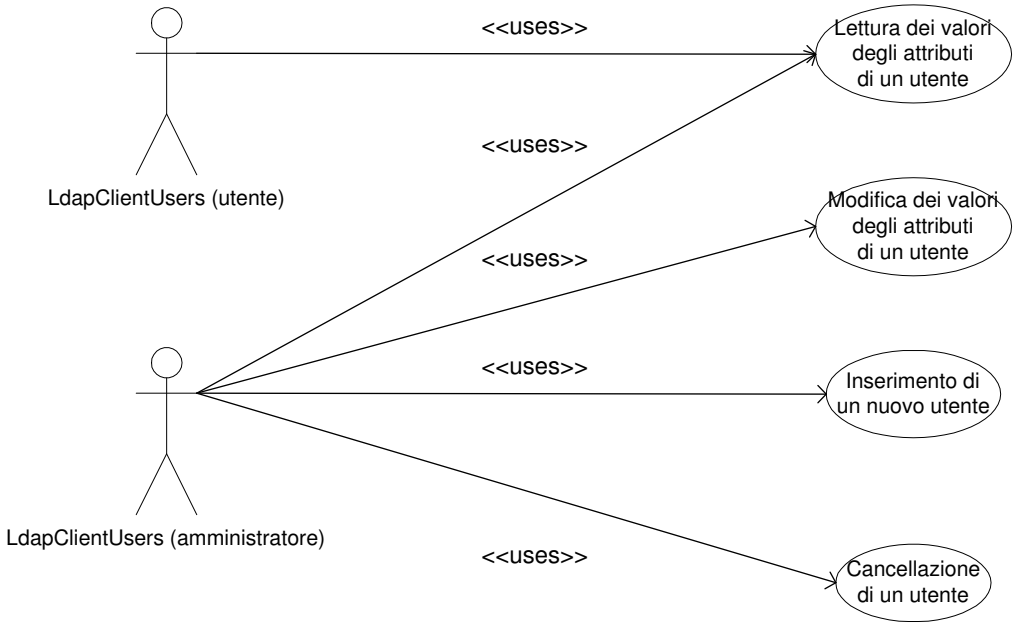

Figura 6.1: Use case diagram per il client LdapClientUsers

All'avvio del client vengono richiesti lo *username* e la *password* per accedere all'EJB. Nel nostro caso possiamo specificare l'utente *utente* o l'utente *amministratore* con le loro relative password. Questi valori vengono passati al costruttore di un nuovo oggetto di tipo *AppCallbackHandler*, la cui classe viene implementata come *static* all'interno della classe LdapClientUsers, ed implementa l'interfaccia *CallbackHandler*.

Quando il *LoginModule* necessita delle informazioni per autenticare il soggetto, usa l'istanza del CallbackHandler. Infatti l'applicazione che implementa l'interfaccia CallbackHandler, passa l'istanza al *LoginContext*, il quale la passa a sua volta al LoginModule.

L'interfaccia LoginContext fornisce i metodi per autenticare i soggetti e ci permette di sviluppare l'applicazione in modo indipendente dalla tecnologia sotto-

stante utilizzata per l'autenticazione.

Il LoginContext consulta una *Configurazione* per determinare il LoginModule da utilizzare, configurato per l'applicazione, nel nostro caso *LdapLoginModule* **§3.1.1**. Al costruttore del LoginContext viene passato l'identificatore, per riferire la Configurazione da utilizzare, nel nostro caso *ldap*, e l'istanza del CallbackHandler.

```
AppCallbackHandler handler = new AppCallbackHandler(name, password);
LoginContext lc = new LoginContext("ldap", handler);
```

I servizi sottostanti possono richiedere differenti tipi di informazioni inviando *Callbacks* al CallbackHandler. Per esempio, se un servizio ha bisogno di conoscere lo username e password per autenticare un utente, usa un *NameCallback* e un *PasswordCallback*. Il CallbackHandler fornirà poi i valori richiesti settandoli nelle Callbacks:

```
public void handle(Callback[] callbacks) throws java.io.IOException, ↩
    UnsupportedCallbackException {
 for (int i = 0; i < callbacks.length; i++) {
  if (callbacks[i] instanceof NameCallback) {
   NameCallback nc = (NameCallback)callbacks[i];
   nc.setName(username);
  }
  else if (callbacks[i] instanceof PasswordCallback) {
   PasswordCallback pc = (PasswordCallback)callbacks[i];
   pc.setPassword(password);
  }
  else {
   throw new UnsupportedCallbackException(callbacks[i], "[ERRORE] Callback ↩
       non riconosciuta");
  }
 }
}
```

Il LoginModule potrà invocare il metodo *handle* specificando i Callbacks NameCallback e PasswordCallbock, per recuperare lo username e la password specificate dall'utente su console.

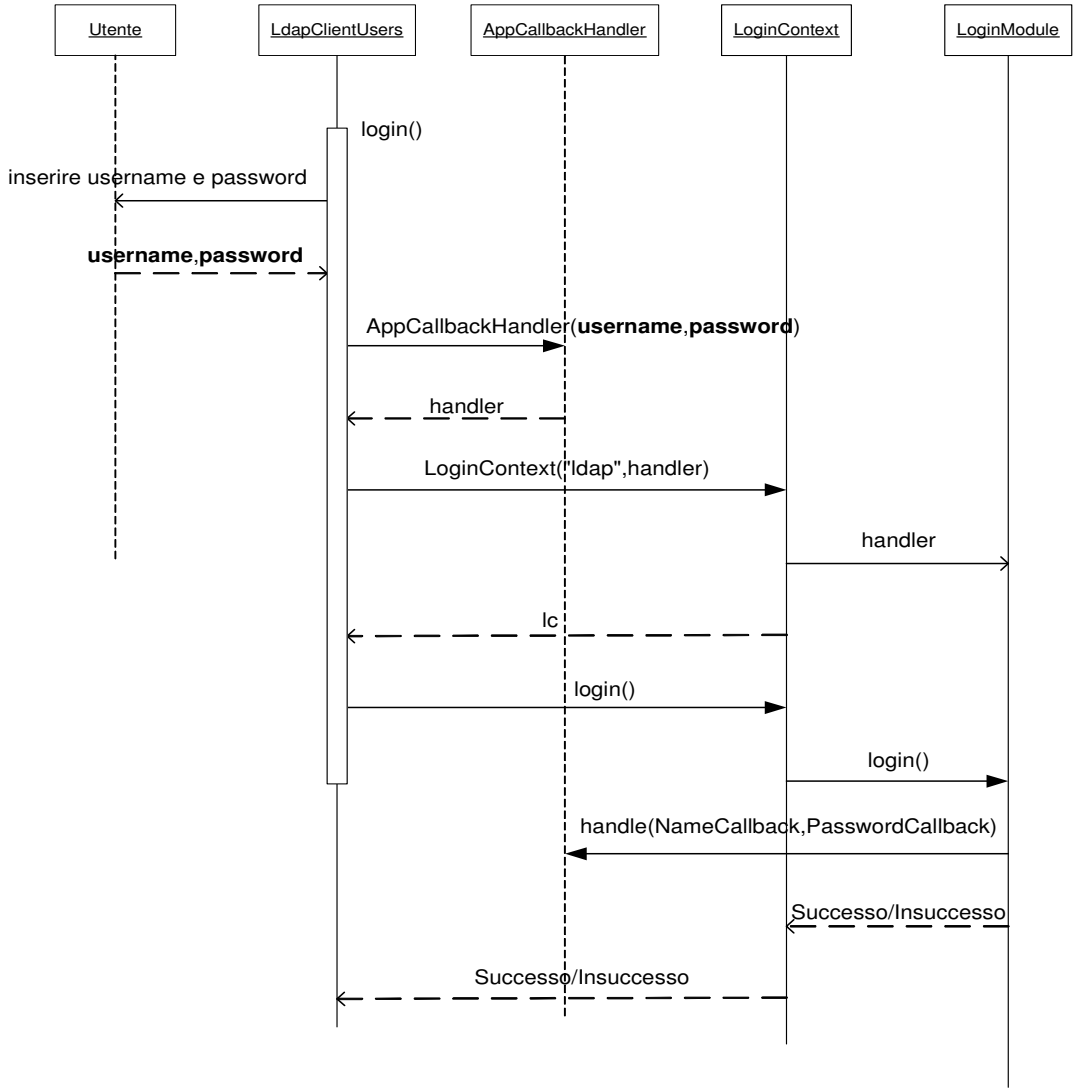

Figura 6.2: Sequence diagram per l'autenticazione nell'accesso all'EJB

L'autenticazione avviene tramite il metodo `login` del LoginContext, il quale invoca a sua volta il metodo `login` del LoginModule specificato nella Confi-

gurazione. Se l'autenticazione ha successo è possibile recuperare l'identità del soggetto:

```
lc.login();
System.out.println(lc.getSubject());
```

Dopo è necessario inizializzare un contesto in cui si specifica come variabili d'ambiente, un nuovo contesto di un servizio di Naming e il suo indirizzo:

```
Hashtable props = new Hashtable();
props.put(InitialContext.INITIAL_CONTEXT_FACTORY,"org.jnp.interfaces. ↩
    NamingContextFactory");
props.put(InitialContext.PROVIDER_URL, "jnp://127.0.0.1:1099");
```

Nella variabile *InitialContext.PROVIDER_URL* si deve inserire l'indirizzo della macchina su cui è in esecuzione l'Application Server JBoss.
Utilizzando il metodo `lookup` del contesto, si ricava il riferimento remoto all'oggetto Home, specificando come parametro il suo nome, che è stato definito nell'interfaccia Home **§4.2**:

```
InitialContext initialContext = new InitialContext(props);
myBean = initialContext.lookup(cmos.LdapServerHome.JNDI_NAME).create();
```

Utilizzando il riferimento remoto all'oggetto Home si invoca il metodo `create` definito sempre nell'interfaccia Home, per ottenere il riferimento remoto all'oggetto di tipo LdapServer, e poter così invocare tutti i metodi necessari dell'EJB.
La classe mette a disposizione i seguenti metodi:

- lettura();
- inserisciUtente();
- modificaUtente();
- cancellaUtente();

- print();

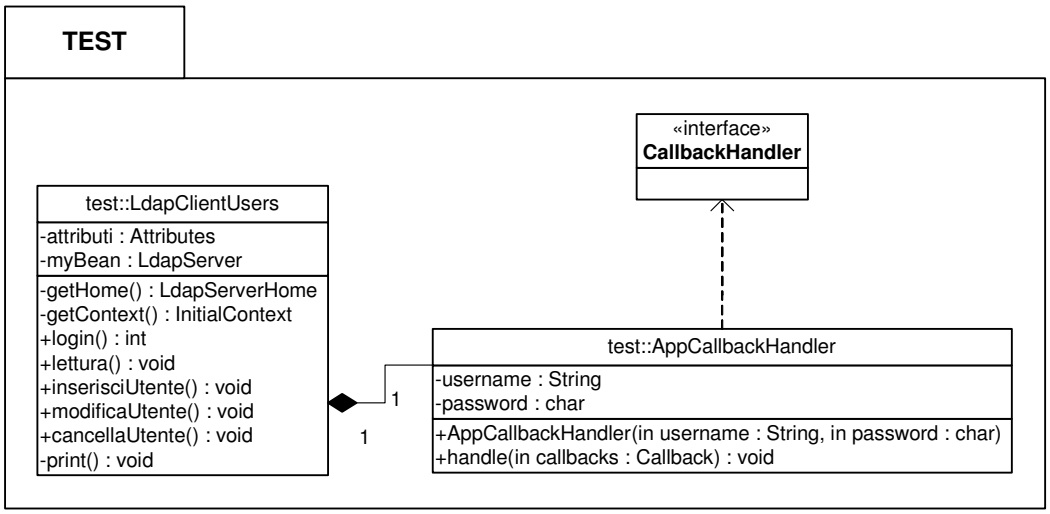

Figura 6.3: Class diagram per la classe LdapClientUsers

6.1.1 Lettura dei valori degli attributi di un utente

Il metodo `lettura` permette di ricavare i valori degli attributi di un utente all'interno del server LDAP, e può essere invocato dagli utenti appartenenti sia al ruolo di *Administrator* che al ruolo di *User*.

Da console viene richiesto lo userid e l'organizationalUnit dell'utente di cui vogliamo ricavare le informazioni. Questi valori sono passati al metodo remoto dell'EJB `query`, il quale ritorna un oggetto di tipo *Attributes* contenente l'insieme delle coppie (nome,valore) degli attributi. Da questo oggetto viene rimosso l'attributo `userPassword`, in modo da non visualizzare il suo valore (anche se cifrato) per ragioni di sicurezza. Viene poi invocato il metodo privato `print` per stampare a video il nome di ciascun attributo e il suo valore.

```
Attributes attributi = myBean.query(userid,ou);
```

```
attributi.remove("userPassword");
print();
```

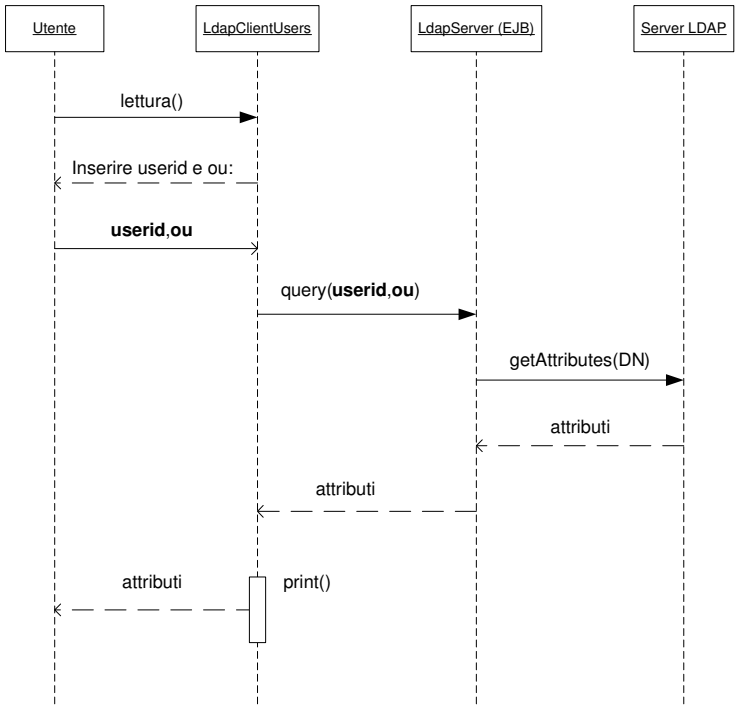

Figura 6.4: Sequence diagram per la lettura dei valori degli attributi di un utente

6.1.2 Inserimento di un nuovo utente

Il metodo `inserisciUtente` permette di inserire un nuovo utente all'interno del server LDAP e può essere invocato solo dall'utente che ha il ruolo di *Administrator*.

Da console viene richiesto il nome dello schema che caratterizza l'insieme degli attributi MUST e MAY per l'utente, nel nostro caso *musica*. Questo valore vie-

ne passato al metodo remoto dell'EJB `schemaQuery`, il quale ritorna un *HashSet* contenente il nome degli attributi MUST e MAY.

```
HashSet valori = myBean.schemaQuery(schema);
```

I nomi degli attributi vengono mostrati sulla console in modo che si possa specificare per ciascuno il valore desiderato. Se non viene specificato alcun valore per gli attributi, il server LDAP produce un errore, in quanto la sintassi specificata nello schema per ciascun attributo, vieta un valore nullo. Per gli attributi `uid` e `userPassword` viene fatto con un ciclo, in modo tale da mostrare continuamente il nome dell'attributo finché l'utente non inserisce un valore. Per gli altri attributi è possibile non specificare un valore da console, ma nel codice è necessario inserire un valore come '-' per evitare un errore sul server LDAP. Questa operazione non è stata fatta per gli attributi `uid` e `userPassword`, perché non inserire un valore, comporterebbe un malfunzionamento del sistema (utente senza userid, senza password...).

```
if(!nome.equals("userPassword") && !nome.equals("uid")) {
 System.out.print("Inserire "+nome+": ");
 valore = br.readLine();
} else {
 valore = "";
 while(valore.equals("")){
  System.out.print("Inserire "+nome+": ");
  valore = br.readLine();
 }
}
if (valore.equals(""))
 valore = "-";
```

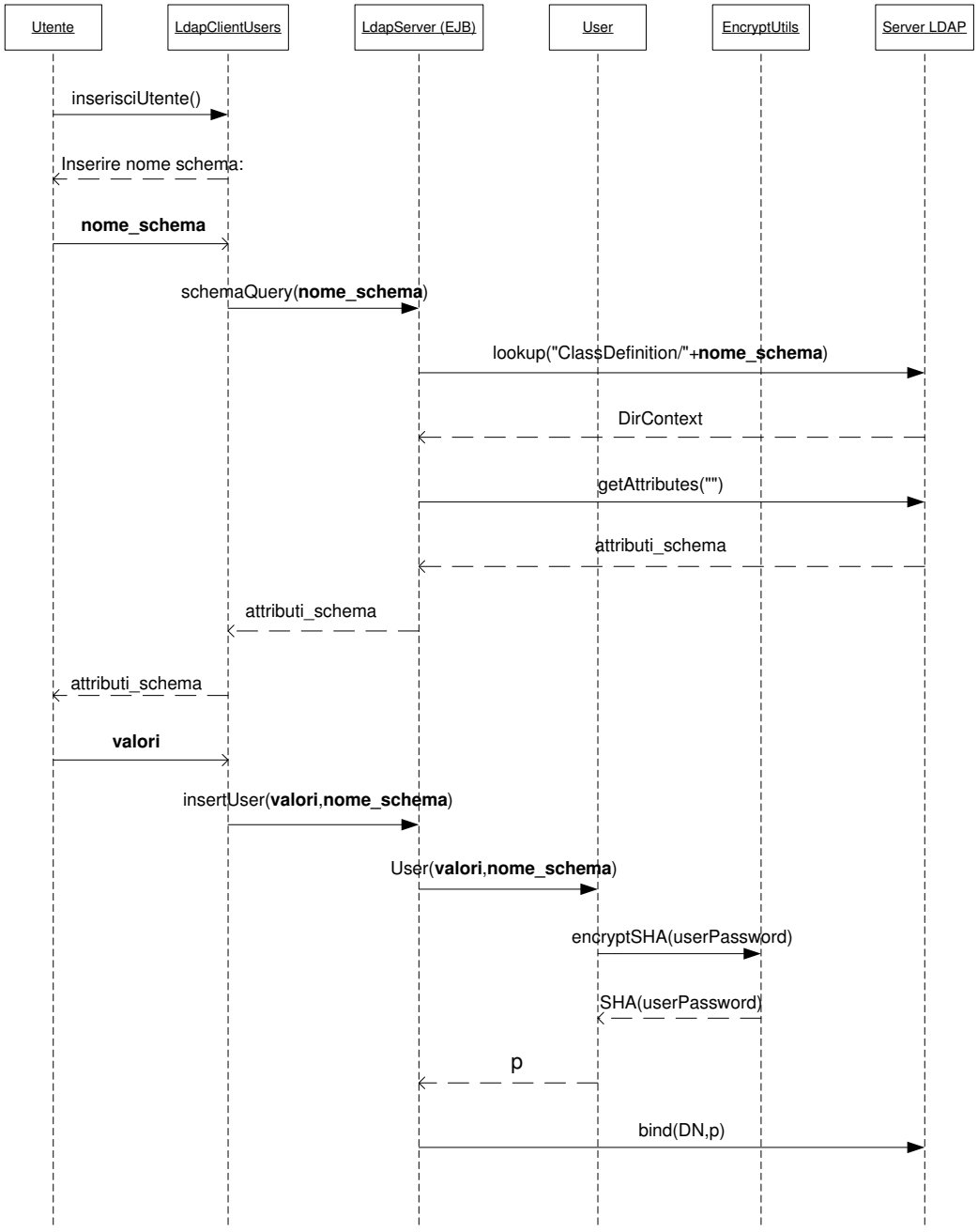

Figura 6.5: Sequence diagram per l'inserimento di un nuovo utente

Infine viene chiamato il metodo remoto `insertUser` dell'EJB per inserire l'utente nel server LDAP, specificando come parametri un *HashMap* creata precedentemente con la lista delle coppie (nome,valore) degli attributi, e il nome dello schema utilizzato.

```
myBean.insertUser(nuoviValori,schema);
```

Per il metodo `inserisciUtente` è stata gestita l'eccezione *RemoteException* che può essere generata nel caso in cui l'utente che si è autenticato per accedere all'EJB, non appartenga al ruolo di Administrator. L'utente verrà avvisato con un messaggio in cui si specifica che non ha il permesso di compiere l'operazione.

6.1.3 Modifica dei valori degli attributi di un utente

Il metodo `modificaUtente` permette di modificare i valori degli attributi di un utente all'interno del server LDAP e può essere invocato solo dall'utente che ha il ruolo di *Administrator*.
Da console viene richiesto lo userid e l'organizationalUnit dell'utente di cui vogliamo compiere le modifiche. Questi valori vengono passati al metodo remoto `query` dell'EJB, che restituisce un oggetto *Attributes* contenente le coppie (nome,valore) per ciascun attributo. Dall'oggetto vengono rimossi gli attributi `objectClass`, `ou` e `uid`. Il primo, in quanto presenta informazioni non utili per l'utente, mentre gli altri non devono essere modificati, per non renderli diversi rispettivamente dall'organizationalUnit e dal Distinguished Name con cui l'utente è stato creato.

```
attributi = myBean.query(userid,ou);
attributi.remove("objectClass");
attributi.remove("ou");
```

```
attributi.remove("uid");
```

I nomi degli attributi vengono visualizzati sulla console insieme al vecchio valore, per permettere all'utente di inserire un nuovo valore. Il valore dell'attributo `userPassword` non viene visualizzato per ragioni di sicurezza. È possibile non specificare un nuovo valore, in quanto viene creato un *HashMap* all'interno del quale vengono inseriti solo gli attributi che presentano un nuovo valore. Infine viene invocato il metodo remoto `modifyUser` dell'EJB, al quale vengono passati come parametri l'HashMap contenenti gli attributi modificati, l'userid e l'organizationalUnit dell'utente di cui vogliamo compiere le modifiche.

```
if (!nuovoValore.equals(""))
 nuoviValori.put(attr.getID(),nuovoValore);
myBean.modifyUser(nuoviValori,userid,ou);
```

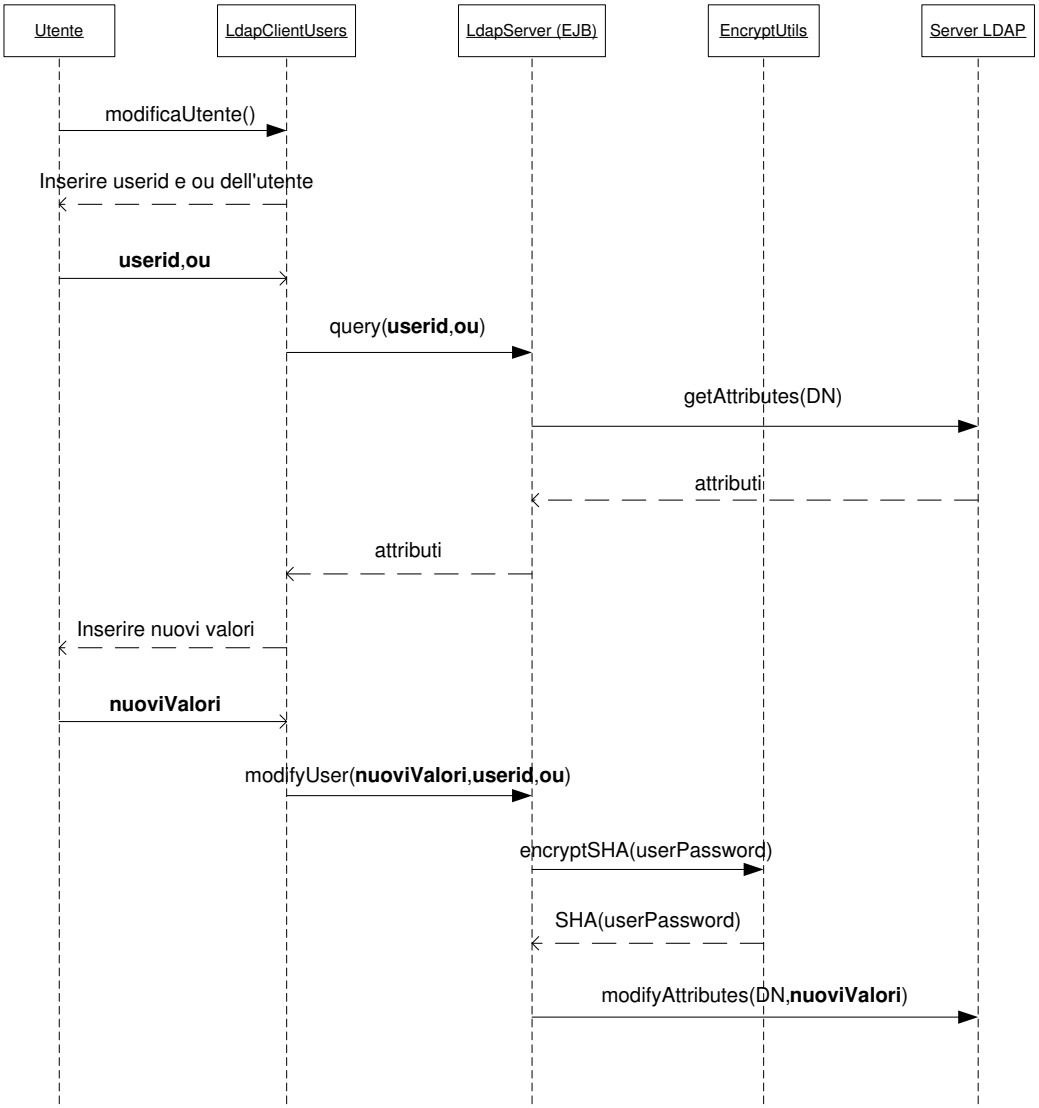

Figura 6.6: Sequence diagram per la modifica dei valori degli attributi dell'utente

6.1.4 Cancellazione di un utente

Il metodo `cancellaUtente` permette di cancellare un utente all'interno del server LDAP e può essere invocato solo dall'utente che ha il ruolo di *Administrator*.

Da console viene richiesto lo userid e l'organizationalUnit dell'utente che vogliamo cancellare. Questi valori vengono passati al metodo remoto `deleteUser` dell'EJB che compie l'operazione effettiva.

```
myBean.deleteUser(userid,ou);
```

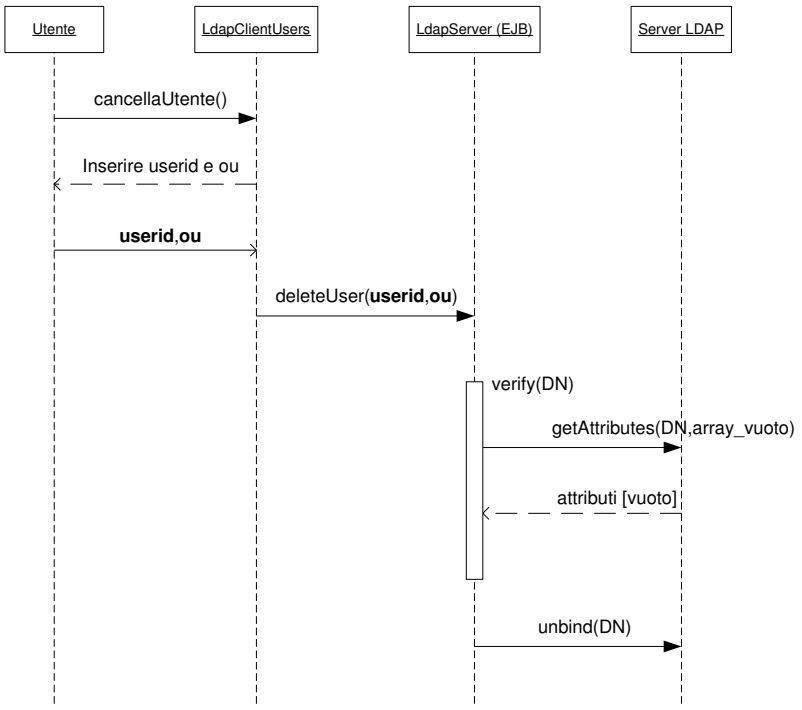

Figura 6.7: Sequence diagram per la cancellazione di un utente

Capitolo 7

Gestione dei ruoli nel server LDAP

7.1 Implementazione del client LdapClientRoles

La classe *LdapClientRoles.java* permette di gestire i ruoli nel server LDAP, offrendo dei metodi che interagiscono con l'EJB *LdapServer*. Le operazioni messe a disposizione dal client sono:

- creazione di un nuovo ruolo;
- cancellazione di un ruolo esistente;
- aggiungere un utente ad un ruolo;
- rimuovere un utente da un ruolo;

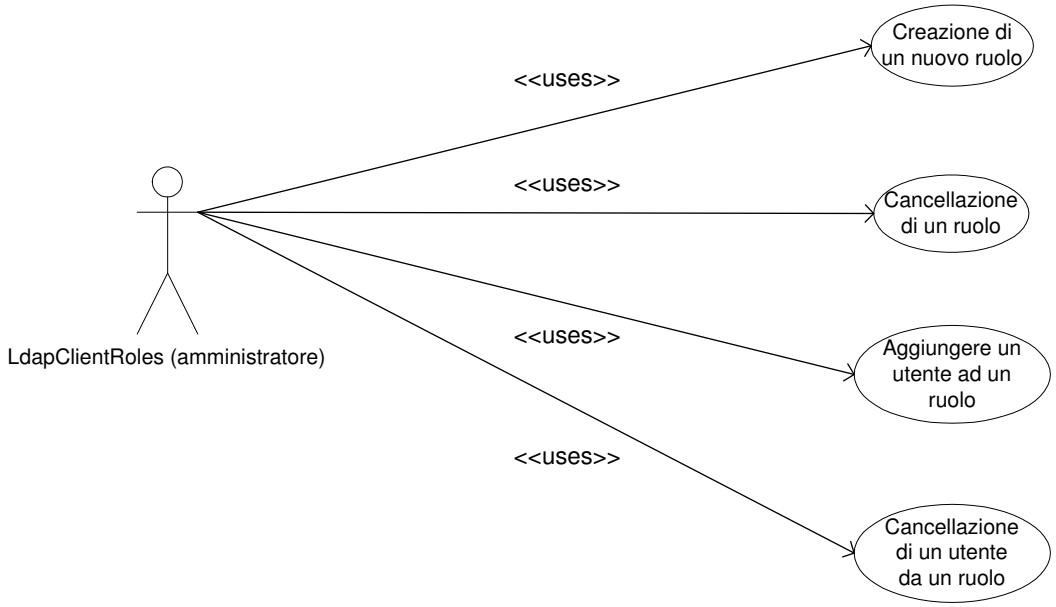

Figura 7.1: Use case diagram per il client LdapClientRoles

All'avvio del client vengono richiesti lo *username* e la *password* per accedere all'EJB. Nel nostro caso possiamo specificare l'utente *utente* o l'utente *amministratore* con le loro relative password. Questi valori vengono passati al costruttore di un nuovo oggetto di tipo *AppCallbackHandler*, la cui classe viene implementata come *static* all'interno della classe LdapClientUsers, ed implementa l'interfaccia *CallbackHandler*.

Quando il *LoginModule* necessita delle informazioni per autenticare il soggetto, usa l'istanza del CallbackHandler. Infatti l'applicazione che implementa l'interfaccia CallbackHandler, passa l'istanza al *LoginContext*, il quale la passa a sua volta al LoginModule.

L'interfaccia LoginContext fornisce i metodi per autenticare i soggetti e ci permette di sviluppare l'applicazione in modo indipendente dalla tecnologia sottostante utilizzata per l'autenticazione.

Il LoginContext consulta una *Configurazione* per determinare il LoginModule da utilizzare, configurato per l'applicazione, nel nostro caso *LdapLoginModule* **§3.1.1**. Al costruttore del LoginContext viene passato l'identificatore, per riferire la Configurazione da utilizzare, nel nostro caso *ldap*, e l'istanza del CallbackHandler.

```
AppCallbackHandler handler = new AppCallbackHandler(name, password);
LoginContext lc = new LoginContext("ldap", handler);
```

I servizi sottostanti possono richiedere differenti tipi di informazioni inviando *Callbacks* al CallbackHandler. Per esempio, se un servizio ha bisogno di conoscere lo username e password per autenticare un utente, usa un *NameCallback* e un *PasswordCallback*. Il CallbackHandler fornirà poi i valori richiesti settandoli nelle Callbacks:

```
public void handle(Callback[] callbacks) throws java.io.IOException, ↩
    UnsupportedCallbackException {
 for (int i = 0; i < callbacks.length; i++) {
  if (callbacks[i] instanceof NameCallback) {
   NameCallback nc = (NameCallback)callbacks[i];
   nc.setName(username);
  }
  else if (callbacks[i] instanceof PasswordCallback) {
   PasswordCallback pc = (PasswordCallback)callbacks[i];
   pc.setPassword(password);
  }
  else {
   throw new UnsupportedCallbackException(callbacks[i], "[ERRORE] Callback ↩
       non riconosciuta");
  }
 }
}
```

Il LoginModule potrà invocare il metodo *handle* specificando i Callbacks NameCallback e PasswordCallbock, per recuperare lo username e la password specificate dall'utente su console.

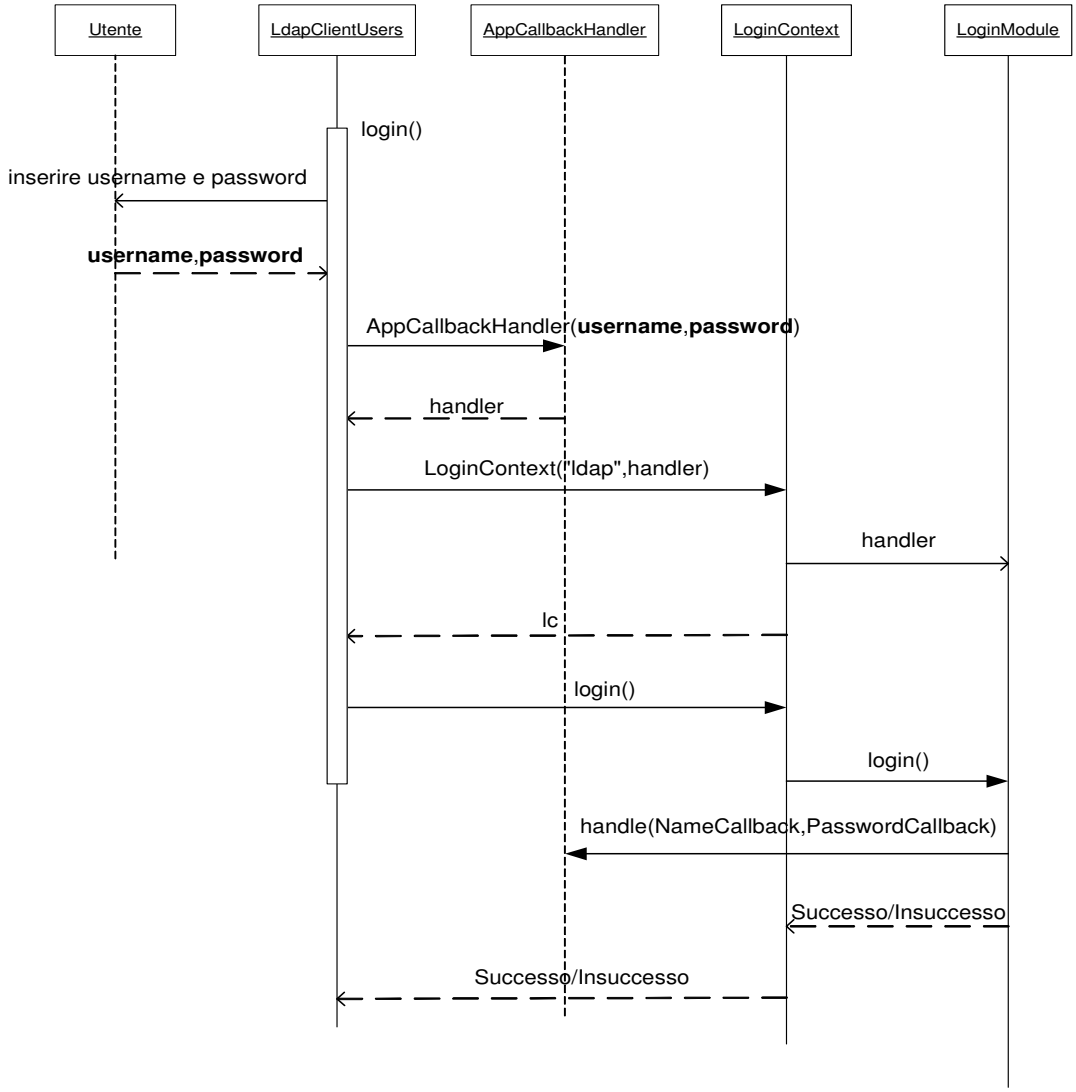

Figura 7.2: Sequence diagram per l'autenticazione nell'accesso all'EJB

L'autenticazione avviene tramite il metodo `login` del LoginContext, il quale invoca a sua volta il metodo `login` del LoginModule specificato nella Confi-

gurazione. Se l'autenticazione ha successo è possibile recuperare l'identità del soggetto:

```
lc.login();
System.out.println(lc.getSubject());
```

Dopo è necessario inizializzare un contesto in cui si specifica come variabili d'ambiente, un nuovo contesto di un servizio di Naming e il suo indirizzo. Queste informazioni non vengono inserite nel codice, ma in un file chiamato *jndi.properties*, in modo da poter cambiare l'indirizzo senza modificare il codice.

```
Hashtable props = new Hashtable();
props.put(InitialContext.INITIAL_CONTEXT_FACTORY,"org.jnp.interfaces. ↩
    NamingContextFactory");
props.put(InitialContext.PROVIDER_URL, "jnp://127.0.0.1:1099");
```

Nella variabile *InitialContext.PROVIDER_URL* si deve inserire l'indirizzo del server in cui viene eseguito l'EJB LdapServer.
Utilizzando il metodo `lookup` del contesto, si ricava il riferimento remoto all'oggetto Home, specificando come parametro il suo nome, che è stato definito nell'interfaccia Home **§4.2**:

```
InitialContext initialContext = new InitialContext(props);
myBean = initialContext.lookup(cmos.LdapServerHome.JNDI_NAME).create();
```

Utilizzando il riferimento remoto all'oggetto Home si invoca il metodo `create` definito sempre nell'interfaccia Home, per ottenere il riferimento remoto all'oggetto di tipo LdapServer, e poter così invocare tutti i metodi necessari dell'EJB.
La classe mette a disposizione i seguenti metodi:

- inserisciRuolo();
- cancellaRuolo();

- aggiungiUtenteAlRuolo();
- cancellaUtenteDalRuolo();

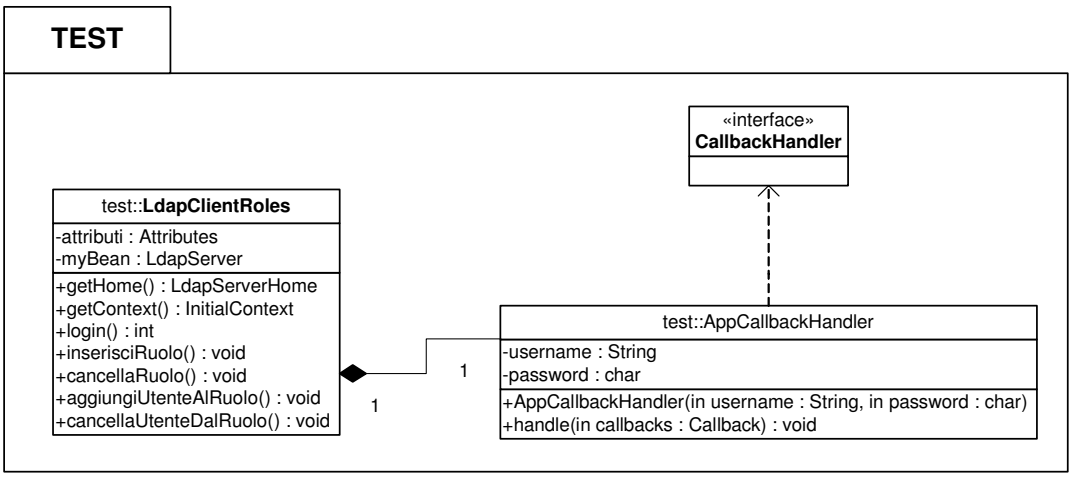

Figura 7.3: Class diagram per la classe LdapClientRoles

7.1.1 Creazione di un nuovo ruolo

Il metodo `inserisciRuolo` permette di creare un nuovo ruolo all'interno del server LDAP e può essere invocato solo dall'utente che ha il ruolo di *Administrator*.

Da console viene richiesto il nome del nuovo ruolo e l'organizationalUnit sotto la quale deve essere creato. Questi valori vengono passati al metodo remoto `insertRole` dell'EJB per compiere l'operazione effettiva.

```
myBean.insertRole(ruolo,ou);
```

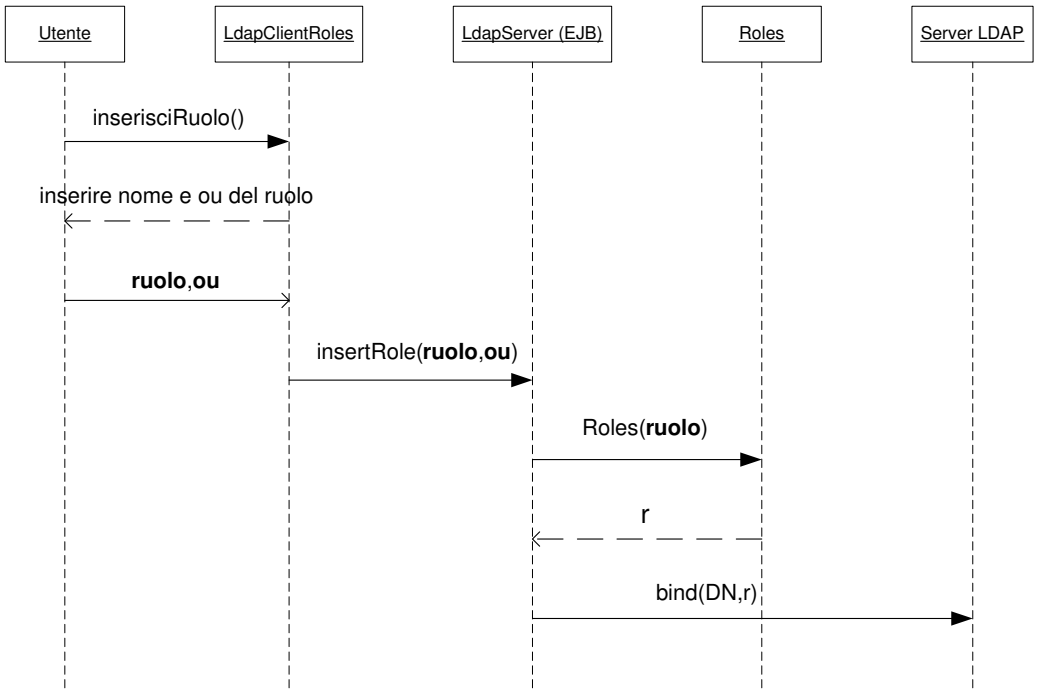

Figura 7.4: Sequence diagram per la creazione di un nuovo ruolo

7.1.2 Cancellazione di un ruolo

Il metodo `cancellaRuolo` permette di cancellare un ruolo esistente all'interno del server LDAP e può essere invocato solo dall'utente che ha il ruolo di *Administrator*.

Da console viene richiesto il nome del ruolo e l'organizationalUnit sotto la quale il ruolo è stato creato. Questi valori vengono passati al metodo remoto `deleteRole` dell'EJB per compiere l'operazione effettiva.

```
myBean.deleteRole(ruolo,ou);
```

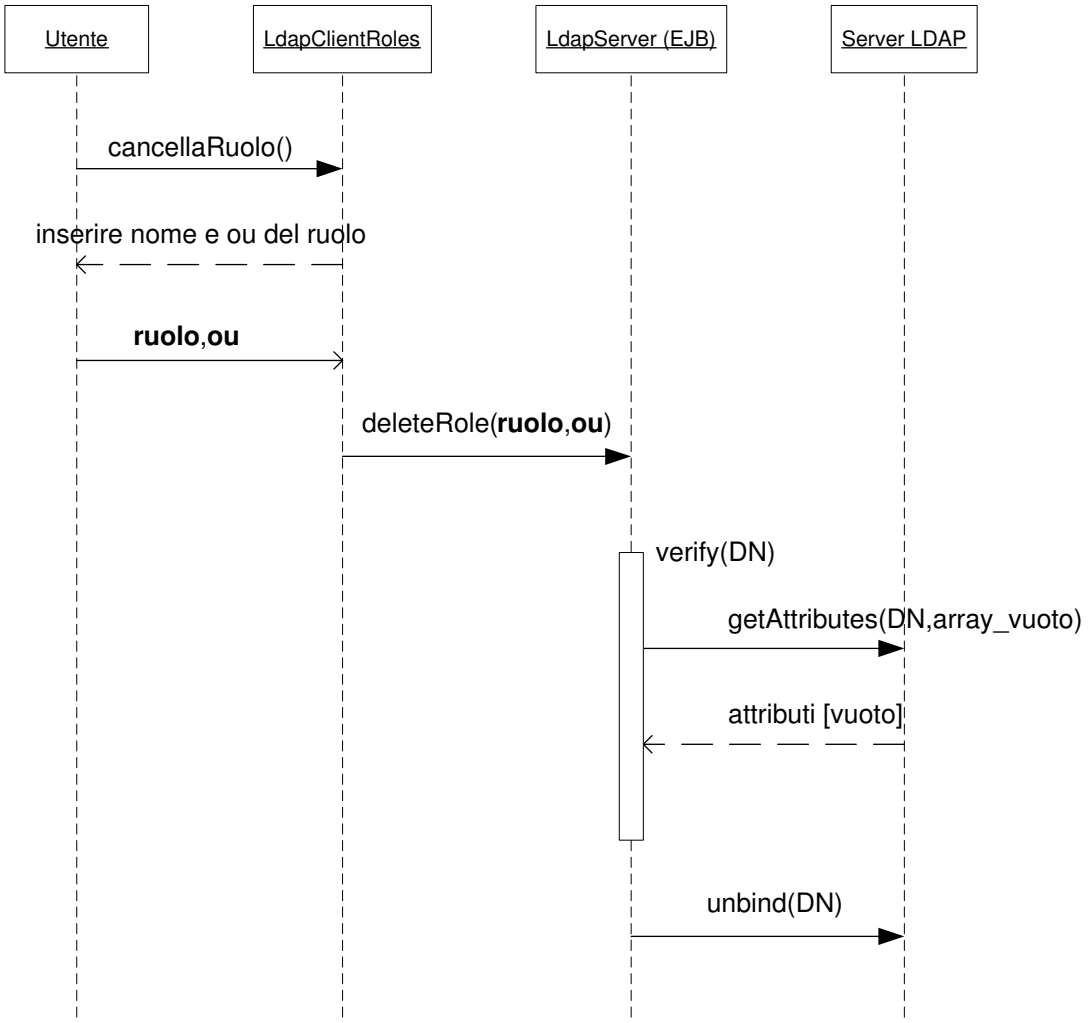

Figura 7.5: Sequence diagram per la cancellazione di un ruolo

7.1.3 Aggiungere un utente ad un ruolo

Il metodo `aggiungiUtenteAlRuolo` permette di aggiungere un utente ad un ruolo all'interno del server LDAP e può essere invocato solo dall'utente che ha il ruolo di *Administrator*.

Da console viene richiesto lo userid e l'organizationalUnit dell'utente, il nome e l'organizationalUnit del ruolo al quale l'utente deve essere aggiunto. Questi valori vengono passati al metodo remoto `addUserToRole` dell'EJB per compiere l'operazione effettiva.

```
myBean.addUserToRole(userid,ouUtente,ruolo,ouRuolo);
```

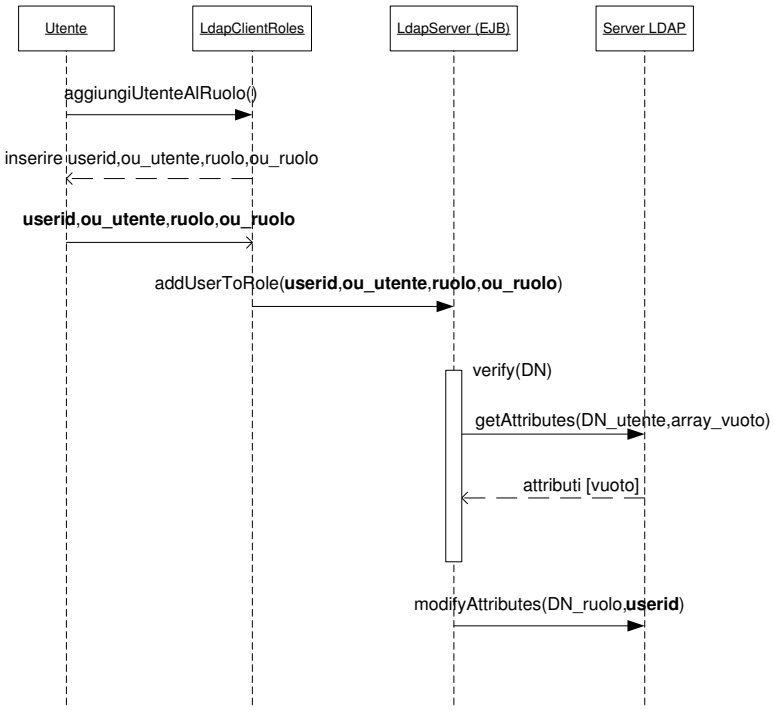

Figura 7.6: Sequence diagram per aggiungere un utente ad un ruolo

7.1.4 Cancellazione di un utente dal ruolo

Il metodo `cancellaUtenteDalRuolo` permette di cancellare un utente da un ruolo all'interno del server LDAP e può essere invocato solo dall'utente che ha il ruolo di *Administrator*.

Da console viene richiesto lo userid dell'utente, il nome e l'organizationalUnit del ruolo al quale l'utente è stato assegnato. Questi valori vengono passati al metodo remoto `delUserFromRole` dell'EJB per compiere l'operazione effettiva. Notare che non è necessario richiedere all'utente l'organizationalUnit dell'utente da cancellare, perché l'operazione consiste solo nel rimuovere dall'entry del ruolo, l'attributo *description*, che assume come valore l'userid dell'utente.

```
myBean.delUserFromRole(userid,ruolo,ou);
```

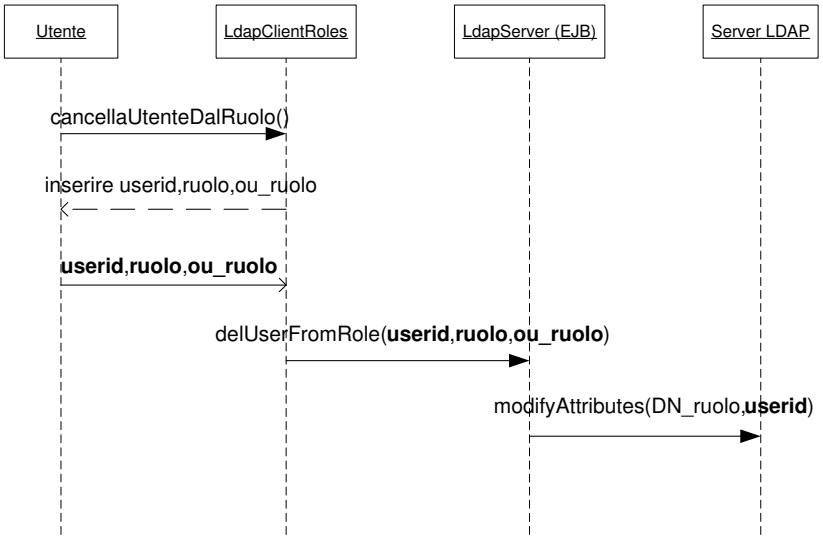

Figura 7.7: Sequence diagram per la cancellazione di un utente da un ruolo

Parte III

Test e conclusioni

Capitolo 8

Test

Sono stati svolti dei test per mostrare l'efficienza del *connection pooling* implementato tra l'Application Server e il server LDAP **§3.1.2**. Il server LDAP è stato installato su una macchina differente da quella su cui è stato installato l'Application Server. I client vengono eseguiti in locale, sulla stessa macchina dell'Application Server.
I test eseguiti sono di due tipi:

1. viene generato un numero di Thread variabile da 10 a 200 con incremento di 10 (quindi 20 test), e ciascun Thread compie una singola richiesta al server LDAP;

2. viene generato un numero di Thread fisso pari a 50, e ciascun Thread compie un numero di richieste al server LDAP variabile da 10 a 200 con incremento di 10 (quindi 20 test);

Questi due test vengono eseguiti senza connection pooling e con connection pooling. Nel primo caso l'inizializzazione e creazione del *contesto ldap* non viene

svolta all'interno del metodo `ejbCreate` dell'EJB **§4.4**, ma viene fatta prima della richiesta al server LDAP. Dopo che la richiesta è stata eseguita, il contesto viene chiuso. Nel secondo caso l'inizializzazione e creazione del contesto ldap avviene all'interno del metodo `ejbCreate` dell'EJB. La chiusura del contesto viene eseguita all'interno del metodo `ejbRemove` dell'EJB, che sarà invocato, solo quando ritenuto necessario dal container.

Ciascun test viene eseguito un numero di volte pari a 15. Ogni test restituisce il tempo di esecuzione totale, dalla generazione dei Thread all'ultima richiesta al server LDAP dell'ultimo Thread. Di questi 15 valori viene fatta la media, e il calcolo dell'intervallo di confidenza del 90% utilizzando il valore che la variabile `t student` assume per `n` pari a 14 (numero iterazioni - 1), cioè 1,761.

Viene inoltre assegnato a ciascun EJB un identificatore univoco, facendo generare nel metodo `ejbCreate` un numero intero random. È stato creato anche un membro statico nella classe che implementa l'EJB, che viene incrementato nel metodo `ejbCreate` in modo tale da funzionare da contatore e ci permette così di conoscere il numero di istanze di EJB create.

Nel caso di connection pooling, ciascun Thread invoca il metodo `query` dell'EJB per fare la richiesta al server LDAP. Questo metodo restituisce al Thread il numero random, cioè l'identificatore dell'istanza di EJB che ha servito la richiesta, e il valore corrente del contatore. Il Thread inserisce l'identificatore in un *HashSet solo se* non è già presente all'interno, in modo tale da poter misurare alla fine di ciascun test il numero di istanze di EJB che sono state utilizzate per servire le richieste di ogni Thread. Nel caso dei test senza connection pooling, ciascun Thread invoca un metodo remoto dell'EJB chiamato `totale`, che

chiama in modo sequenziale i metodi dell'EJB:

- `init`: per l'inizializzazione e creazione del contesto ldap;
- `query`: per la richiesta al server LDAP;
- `close`: per la chiusura del contesto ldap;

Anche in questo caso il metodo `totale` restituisce al Thread l'identificatore dell'EJB che ha eseguito la richiesta e il contatore.
Alla fine di un test, vengono recuperate le seguenti informazioni:

- tempo di esecuzione;
- numero di istanze di EJB invocate durante tutta la durata del test;
- numero di istanze di EJB create dall'Application Server;

In figura **8.1** sono riportati i risultati dei test con e senza connection pooling al variare del numero dei Thread generati.

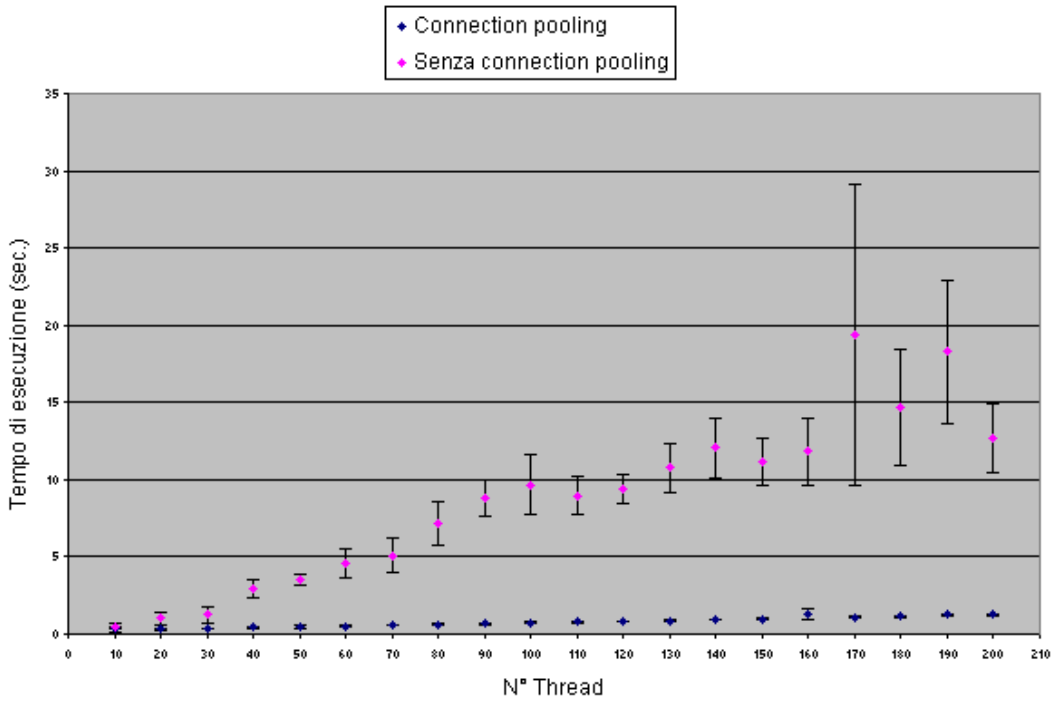

Figura 8.1: Test al variare del numero dei Thread

Come è possibile vedere dalla figura, l'andamento non è sempre regolare. Questo perché ciascun valore è il risultato di una media di 15 valori, quindi alcuni dei test hanno prodotto un tempo di esecuzione molto alto. Il motivo per cui si verifica questo fenomeno, può essere ricondotto al grosso numero di istanze di EJB creato dall'Application Server JBoss per supportare l'aumento delle richieste dovuto all'incremento del numero dei Thread. La creazione di una nuova istanza comporta la chiamata del metodo `ejbCreate`, all'interno del quale viene creato un nuovo contesto *DirContext* per comunicare con il server LDAP e stabilire una connessione SSL. L'operazione è quindi molto costosa da un punto di vista dei tempi, e può compromettere il valore finale. Il fenomeno evidenziato nella figura, è meno evidente se i vari componenti dell'architettura

del sistema vengono installati su macchine differenti. Questo perché il carico delle operazioni viene distribuito su più macchine, diminuendo così la probabilità che l'Application Server JBoss sia costretto a creare nuove istanze di EJB, dovuto al fatto che quest'ultime sono tutte occupate nell'interazione con il server LDAP. Nelle figure **8.2** e **8.3** viene mostrato l'andamento del numero delle richieste di istanze EJB durante l'esecuzione dei test, sia con connection pooling che senza.

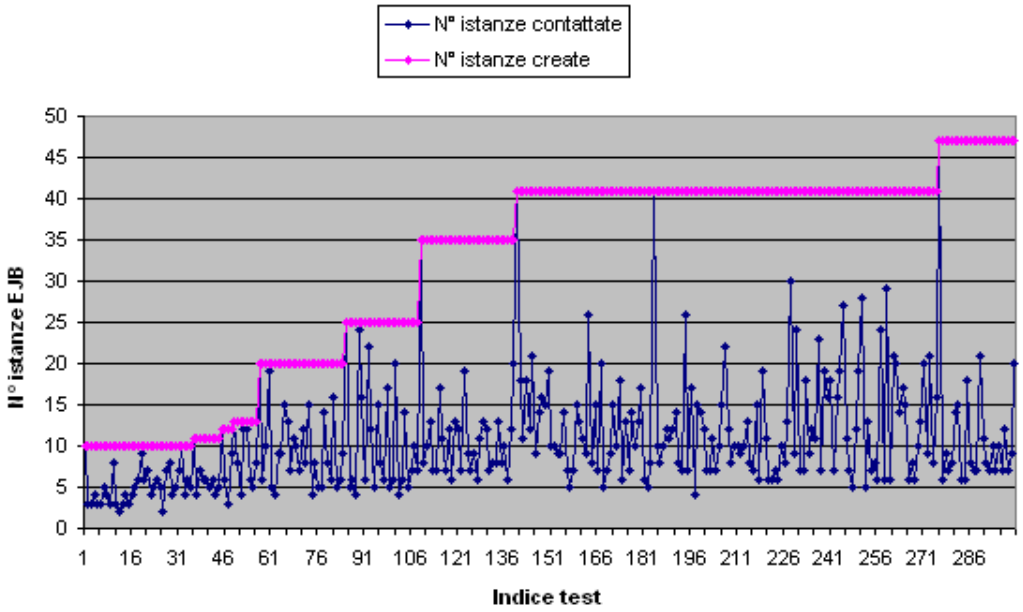

Figura 8.2: Numero istanze contattate e create durante l'esecuzione del primo tipo di test con connection pooling

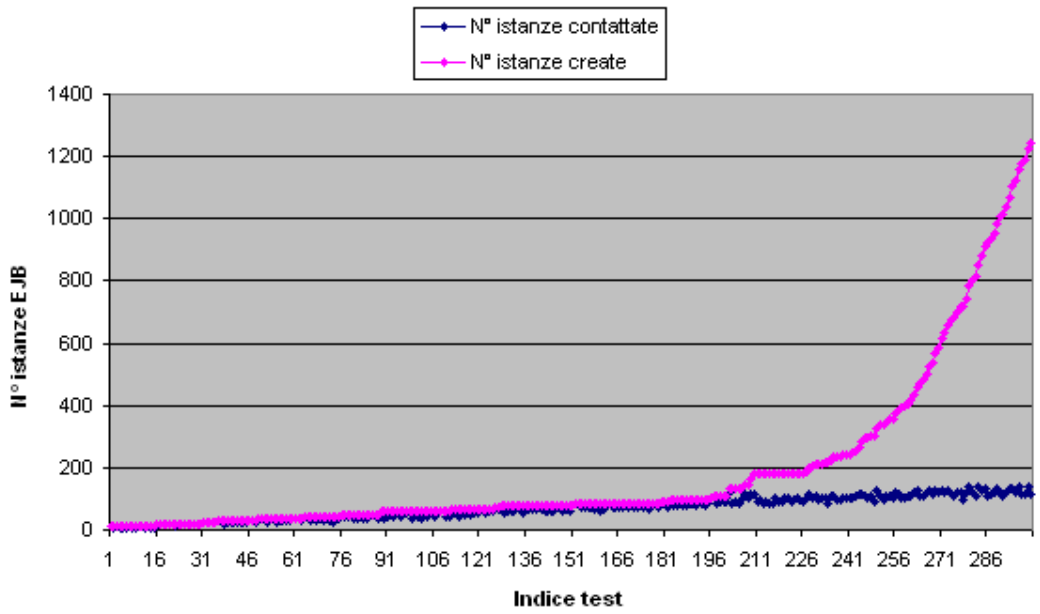

Figura 8.3: Numero istanze contattate e create durante l'esecuzione del primo tipo di test senza connection pooling

L'utilizzo del connection pooling mostra dei tempi di esecuzione molto più bassi rispetto ai test in cui non è stato utilizzato. Questo perché senza l'utilizzo del connection pooling, ogni richiesta al server LDAP comporta la creazione del contesto, il tempo per ottenere i dati dal server LDAP e la chiusura del contesto. Le operazioni di creazione e chiusura del contesto sono molto costose da un punto di vista dei tempi. Infatti nel connection pooling l'operazione di creazione del contesto viene fatta una sola volta all'interno del metodo `ejbCreate`. Il contesto verrà poi condiviso tra tutti i client che faranno richiesta all'EJB.
In figura **8.4** sono riportati i risultati dei test con e senza connection pooling al variare del numero delle richieste al server LDAP.

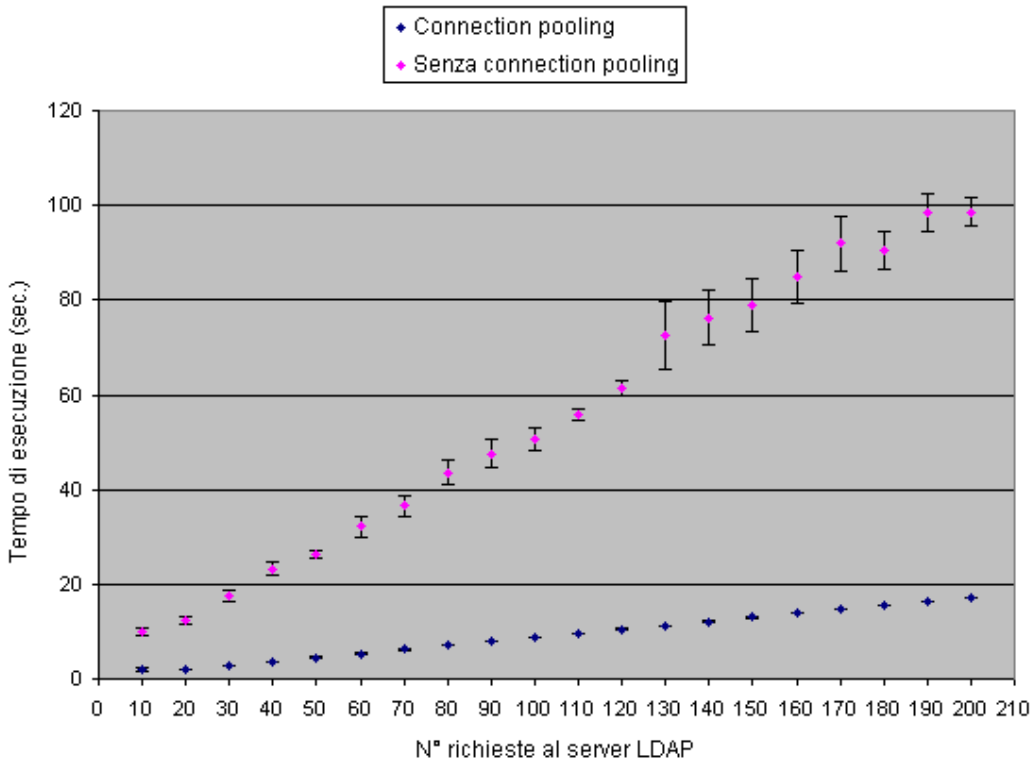

Figura 8.4: Test al variare del numero delle richieste al server LDAP

In questo tipo di test, il numero delle istanze di EJB create è fisso a 50, pari al numero dei Thread. Importante notare che questo numero di istanze non aumenta all'aumentare del numero delle richieste al server LDAP, ma rimane costante al valore di 50. Questo perché il numero di istanze di EJB create è sufficiente a soddisfare il numero di richieste al server LDAP. Questo comportamento si verifica sia con il connection pooling che senza, come viene mostrato nelle figure **8.5** e **8.6**, anche se senza il suo utilizzo, i tempi di esecuzione sono molto più elevati, proprio perché ogni richiesta è preceduta dalla creazione del contesto e seguita dalla chiusura dello stesso.

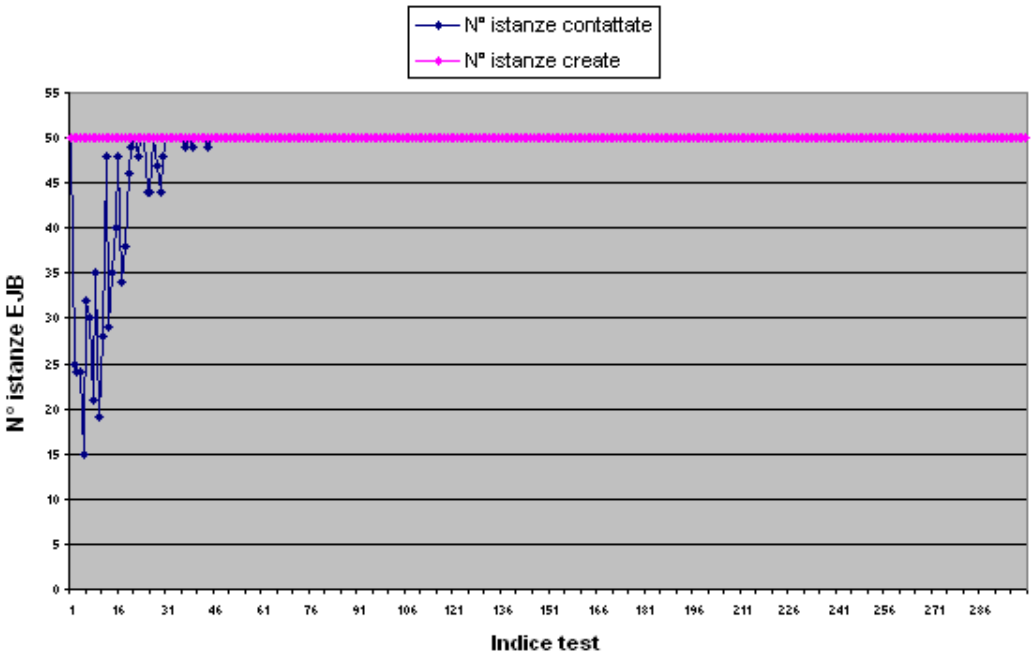

Figura 8.5: Numero istanze contattate e create durante l'esecuzione del secondo tipo di test con connection pooling

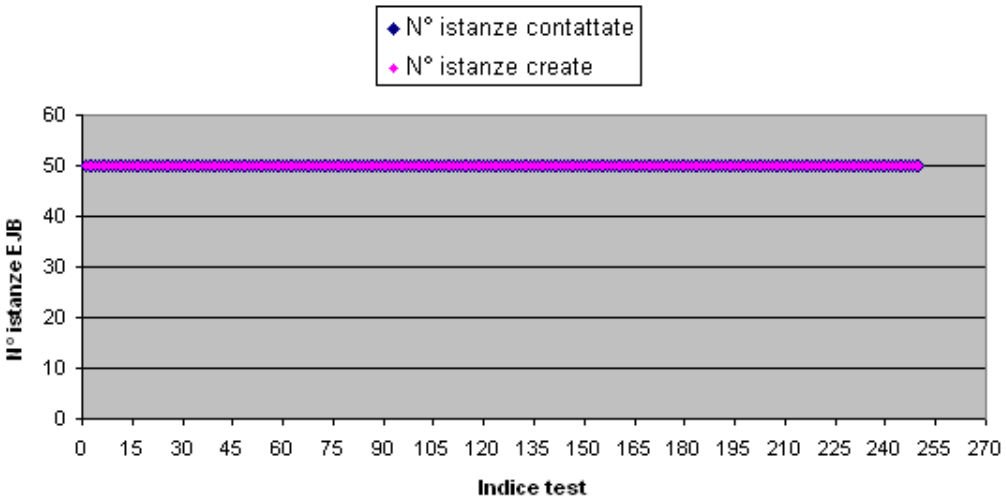

Figura 8.6: Numero istanze contattate e create durante l'esecuzione del secondo tipo di test senza connection pooling

I test sono stati svolti con le seguenti opzioni nel file *standardjboss.xml*, per la configurazione del connection pooling per lo Stateless Session Bean:

```
<container-pool-conf>
 <MaximumSize>100</MaximumSize>
</container-pool-conf>
```

Capitolo 9

Conclusioni

In questa tesi è stato progettato e implementato un sistema di gestione degli utenti, per un controllo degli accessi basato su attributi. Il sistema è stato implementato utilizzando le seguenti tecnologie e strumenti:

- Java 2 Standard Edition;
- Application Server JBoss;
- OpenLDAP;
- BerkeleyDB;
- OpenSSL;

Nella prima parte del lavoro è stato necessario documentarsi sulla tecnologia *J2EE*, e sui vari componenti da installare, in particolare sulla loro configurazione.
Il sistema mette a disposizione tre client che permettono all'utente di interagire con il server LDAP. Il servizio viene implementato tramite un *Enterprise*

Java Bean (*EJB*) che mette a disposizione i suoi metodi remoti ai client *LdapClientUsers* e *LdapClientRoles*. Il client *LdapClientContext* è stato realizzato come client stand-alone, ed interagisce direttamente col server LDAP, senza il supporto dell'Application Server JBoss.
Il client *LdapClientUsers* permette di:

- creare un nuovo utente;
- cancellare un utente esistente;
- recuperare i valori degli attributi dell'utente;
- modificare i valori degli attributi dell'utente;

Il client *LdapClientRoles* permette di:

- creare un nuovo ruolo;
- cancellare un ruolo esistente;
- aggiungere un utente ad un ruolo;
- cancellare un utente da un ruolo;

Il client *LdapClientContext* permette di:

- creare il contesto root;
- creare un subContext;
- cancellare un subContext;

Attraverso la tecnologia *Java Authentication and Authorization Service* (*JAAS*), è stato realizzato un controllo degli accessi all'EJB, basato sul ruolo dell'utente. È possibile così creare diversi ruoli all'interno del sistema, assegnare gli utenti ai ruoli desiderati e decidere per ciascun ruolo, quali operazioni sono permesse all'utente.
Il metodo dell'EJB per ricavare i valori degli attributi dell'utente, viene messo a disposizione di un controllo degli accessi (Progetto CMOS tecnologia *XACML*) basato sui valori che gli attributi assumono.
Il sistema presenta attualmente i seguenti *limiti*:

- i client *LdapClientUsers* e *LdapClientRoles* funzionano correttamente con un *Directory Information Tree* a un solo livello;
- la password da inserire su console è in chiaro;
- la connessione tra i client *LdapClientUsers*, *LdapClientRoles* e l'Application Server non è protetta da un canale sicuro come SSL;

Questi limiti possono essere superati nei seguenti *sviluppi futuri*:

- interfaccia grafica per l'utente;
- modificare i metodi dell'EJB per funzionare correttamente con un Directory Information Tree a più livelli;
- stabilire un canale SSL tra il client e l'Application Server in modo da rendere sicura la connessione;
- nel caso di una futura implementazione di un *connettore JCA* per

OpenLDAP, utilizzare e sperimentare questa soluzione per il *connection pooling* tra JBoss e OpenLDAP;

Bibliografia

[1] The J2EE 1.4 Tutorial,
http://java.sun.com/j2ee/1.4/docs/tutorial/doc/index.html

[2] EJB Design Patterns,
http://www.theserverside.com/books/wiley/EJBDesignPatterns/index.tss

[3] F. Marinescu. EJB Design Patterns: Advanced Patterns, Processes, and Idioms. Wiley Computer Publishing

[4] Your guide to The JNDI Tutorial,
http://java.sun.com/products/jndi/tutorial/trailmap.html

[5] OpenLDAP Administrator's Guide,
http://www.openldap.org/doc/admin22/

[6] LDAP Schema,
http://www.it.ufl.edu/projects/directory/ldap-schema/objectclasses.html

[7] J. Hodges, R. Morgan. rfc3377: Lightweight Directory Access Protocol (v3): Technical Specification. September 2002

[8] M. Wahl, T. Howes, S. Kille. rfc2251: Lightweight Directory Access Protocol (v3). December 1997

[9] M. Wahl, S. Kille, T. Howes. rfc2253: Lightweight Directory Access Protocol (v3): UTF-8 String Representation of Distinguished Names. December 1997

[10] T. Howes, M. Smith. rfc2255: The LDAP URL Format. December 1997

[11] M. Wahl. A Summary of the X.500(96) User Schema for use with LDAPv3. December 1997

[12] J. Hodges, R. Morgan, M. Wahl. rfc2830: Lightweight Directory Access Protocol (v3): Extension for Transport Layer Security

[13] M. Smith. rfc2798: Definition of the inetOrgPerson LDAP Object Class. April 2000

[14] M. Wahl, H. Alvestrand, J. Hodges. rfc2829: Authentication Methods for LDAP. May 2000

[15] M. Wahl, A. Coulbeck, T. Howes, S. Kille. rfc2252: Lightweight Directory Access Protocol (v3): Attribute Syntax Definitions. December 1997

[16] Tutorial for building J2EE Applications using JBOSS and ECLIPSE, http://www.tusc.com.au/tutorial/html/

[17] OpenLDAP SSL/TLS How-To, http://www.openldap.org/pub/ksoper/OpenLDAP_TLS_howto.html

[18] Integrate security infrastructures with JBossSX,
http://www.javaworld.com/javaworld/jw-08-2001/jw-0831-jaas.html

[19] JAAS Based Security in JBoss,
http://www.jboss.ru/translate/jboss24/ch11s78.html